Project Report on

Handwritten Digit Recognition Using Neural Network

Submitted by

Soumya Dey

Roll: **704**

Semester: **VI**

Ramakrishna Mission Vivekananda Centenary College

July 20, 2021

Table of Contents

Abstract

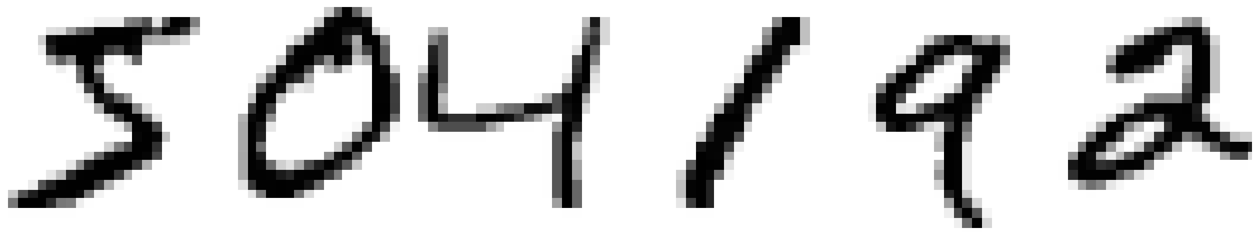
In this report, a neural network is developed from scratch to classify handwritten digit images into the respective digit classes. For building the neural network, the python library named Numpy is used.

Some other python libraries such as Matplotlib, Seaborn, etc. are used for plotting the graphs are visualizing them.

The experiments are performed on the well-known MNIST database to train and validate the model. Then the performance of the model is observed on some new test data.

Introduction

For us humans, it is very easy to recognize some images that we see through our eyes. For millions of years, we have adapted to understand the visual world. Recognizing handwritten digits isn't easy. Rather, we humans are stupendously, astoundingly good at making sense of what our eyes sho



Most people effortlessly recognize those digits as 504192. The difficulty of visual pattern recognition becomes apparent if you attempt to write a computer program to recognize digits like those above.

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 that can learn from those training examples. In other words, the neural network uses the examples to automatically infer rules for recognizing handwritten digits. Furthermore, by increasing the number of training examples, the network can learn more about handwriting, and so improve its accuracy.

We'll use the MNIST data set, which contains tens of thousands of scanned images of handwritten digits, together with their correct classifications. MNIST's name comes from the fact that it is a modified subset of two data sets collected by NIST, the United States National Institute of Standards and Technology.

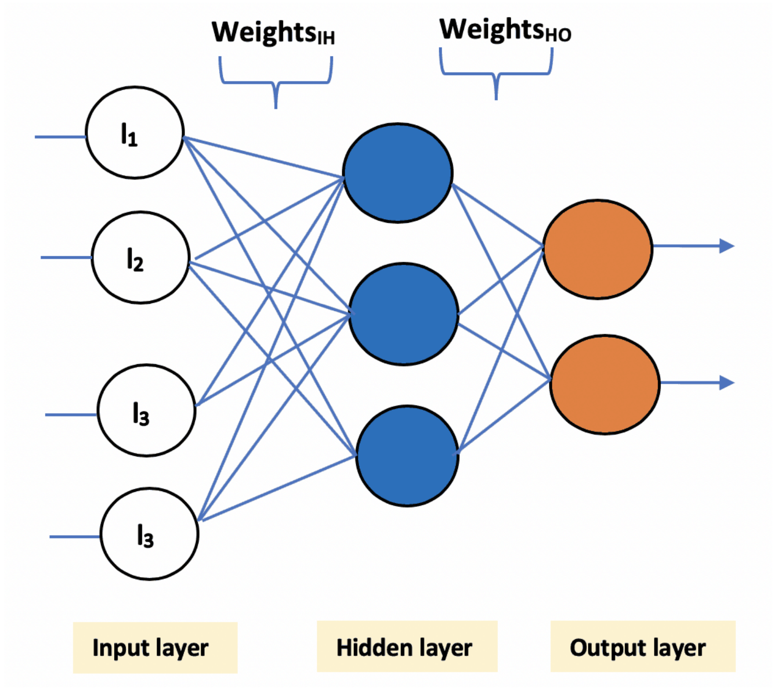
Background

An artificial neural network is a supervised learning algorithm that leverages a mix of multiple hyper-parameters that help in approximating complex relations between input and output. These artificial networks may be used for predictive modeling, adaptive control, and applications where they can be trained via a dataset.

Some of the hyper-parameters in the artificial neural networks include the following:

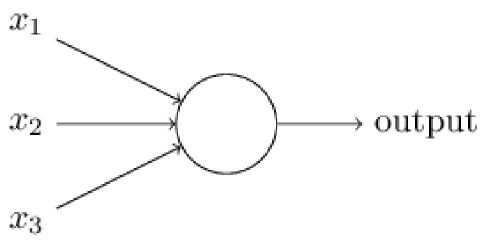
- ➔ Number of hidden layers
- ➔ Number of hidden units
- ➔ Activation function
- ➔ Learning rate

Following is a simplified view of an artificial neural network with a single hidden layer.



Perceptron

A perceptron takes several binary inputs, x1,x2,x3,…, and produces a single binary output. In this example shown the perceptron has three inputs, x1,x2,x3.



The neuron's output, 0 or 1, is determined by whether the weighted sum $\sum_j w_j x_j$ is less than or greater than some *threshold value*. Just like the weights, a threshold is a real number which is a parameter of the neuron. To put it in more precise algebraic terms:

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

Sigmoid Neurons

Just like a perceptron, the sigmoid neuron has inputs, x1,x2,x3,…. But instead of being just 0 or 1, these inputs can also take on any values between 0 and 1.

Also just like a perceptron, the sigmoid neuron has weights for each input, w1,w2,…, and an overall bias, b. But the output is not 0 or 1. Instead, it's σ(w⋅x+b), where σ is called the sigmoid function and is defined by:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

when z=w·x+b is large and positive, the output from the sigmoid neuron is approximately 1. On the other hand, when z=w·x+b is very negative, the output from the sigmoid neuron is approximately 0.

Hence, the behavior of a sigmoid neuron closely approximates a perceptron. It's only when w·x+b is of modest size that there's a deviation from the perceptron model.

Cost Function

Cost Function quantifies the error between predicted values and expected values and presents it in the form of a single real number. Following is our definition of the cost function:

$$C(w, b) = \frac{1}{2n} \sum_{x} || \, y(x) - a \, ||^2$$

Here, w denotes the collection of all weights in the network, b all the biases, n is the total number of training inputs, a is the vector of outputs from the network when x is input, and the sum is over all training inputs, x. This type of C, we will call the quadratic cost function; it’s also sometimes known as the mean squared error or just MSE.

Gradient Descent

Gradient descent is an optimization algorithm. It’s based on a convex function and tweaks its parameters iteratively to minimize a given function (generally Cost function) to its local minimum.

For this project, we are going to use *Stochastic gradient descent*, which approximates the true gradient of C(w, b) by considering a small batch of training samples at a time.

Backpropagation Algorithm

The backpropagation equations provide us with a way of computing the gradient of the cost function.

1. **Input x:** Set the corresponding activation a^1 for the input layer.
2. **Feedforward:** For each $l = 1, 2, 3, \ldots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$
3. **Output error** δ^L**:** Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$
4. **Backpropagate the error:** For each $l = L - 1, L - 2, \ldots, 2$ compute -
   - $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^L)$
5. **Output**: The gradient of the cost function is given by -

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \quad \text{and} \quad \frac{\partial C}{\partial b^l_j} = \delta^l_j$$

Objective

Our task is to download the MNIST handwritten digits dataset and split the data into training and test sets. It is a dataset of 60,000 small square 28×28 pixel grayscale images of handwritten single digits between 0 and 9. It also consists 10,000 images as testing examples for testing the model trained on the 60,000 images.

After the split, we need to train the training dataset with a neural network. The neural network should have 3 hidden layers with 64, 32, and 16 neurons (from left to right).
We will be using the Stochastic Gradient Descent algorithm.

Then we need to test the model on the testing dataset. We have to include *training and testing accuracy* along with *recall, precision, and f1-score* for each class.
The project should also have the plot to show how our training error has reduced for each epoch while training. Additionally, there will be a confusion matrix to understand the performance of the model better.

Result and Analysis

Hyperparameters used for training the neural network:

1. No. of Epochs: **25** 2. Mini batch size: **64** 3. Learning rate: **5.0**

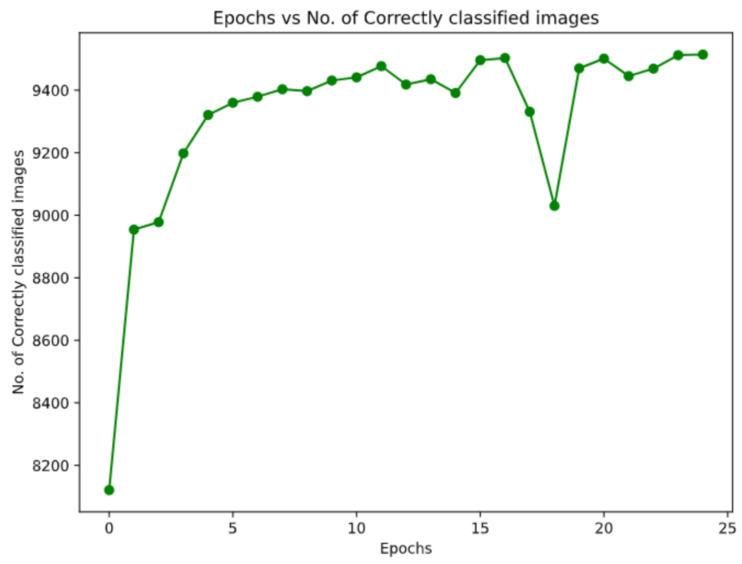
The result after training:

On the final epoch, the network correctly identifies *9514* samples out of *10000* samples.

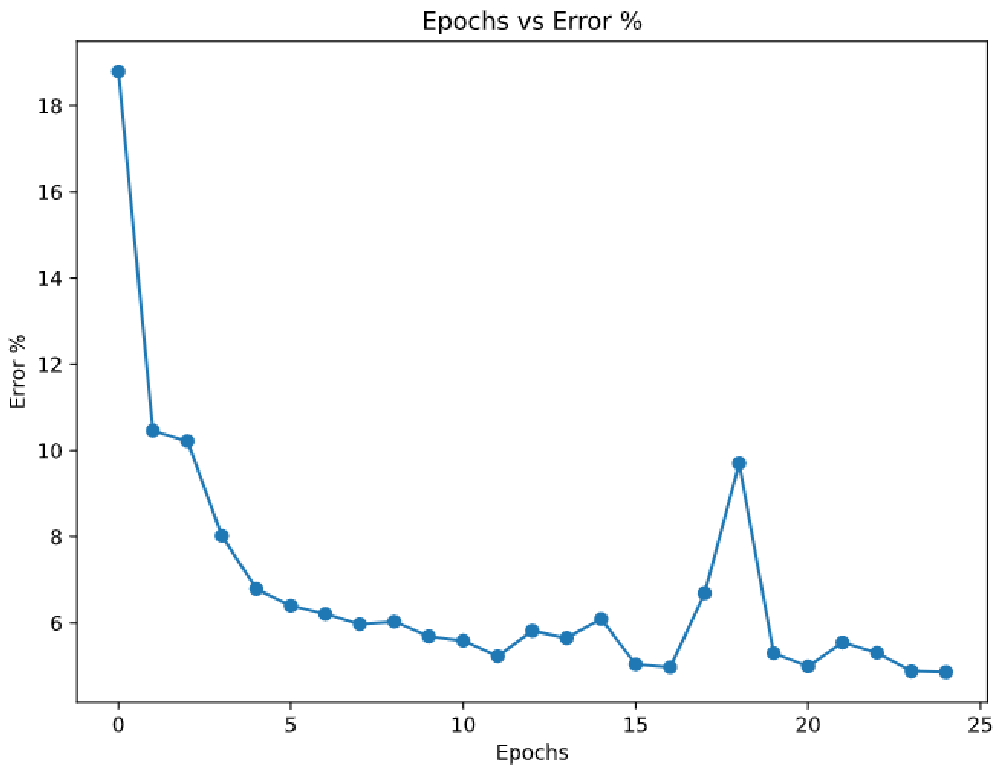
- Training accuracy: **95.14%**

```
net = network.Network([784, 64, 32, 16, 10])
net.SGD(training_data, 25, 64, 5.0, test_data=validation_data)
---------------------------------------------------------------------------
Epoch 0 : 8121 / 10000
Epoch 1 : 8954 / 10000
Epoch 2 : 8978 / 10000
Epoch 3 : 9198 / 10000
Epoch 4 : 9321 / 10000
Epoch 5 : 9360 / 10000
Epoch 6 : 9379 / 10000
Epoch 7 : 9403 / 10000
Epoch 8 : 9397 / 10000
Epoch 9 : 9431 / 10000
Epoch 10 : 9441 / 10000
Epoch 11 : 9477 / 10000
Epoch 12 : 9418 / 10000
Epoch 13 : 9435 / 10000
Epoch 14 : 9391 / 10000
Epoch 15 : 9496 / 10000
Epoch 16 : 9503 / 10000
Epoch 17 : 9331 / 10000
Epoch 18 : 9030 / 10000
Epoch 19 : 9470 / 10000
Epoch 20 : 9501 / 10000
Epoch 21 : 9445 / 10000
Epoch 22 : 9469 / 10000
Epoch 23 : 9512 / 10000
Epoch 24 : 9514 / 10000
```

Plot on No. of Correctly classified images in each epoch:



Plot on No. of Correctly classified images in each epoch:



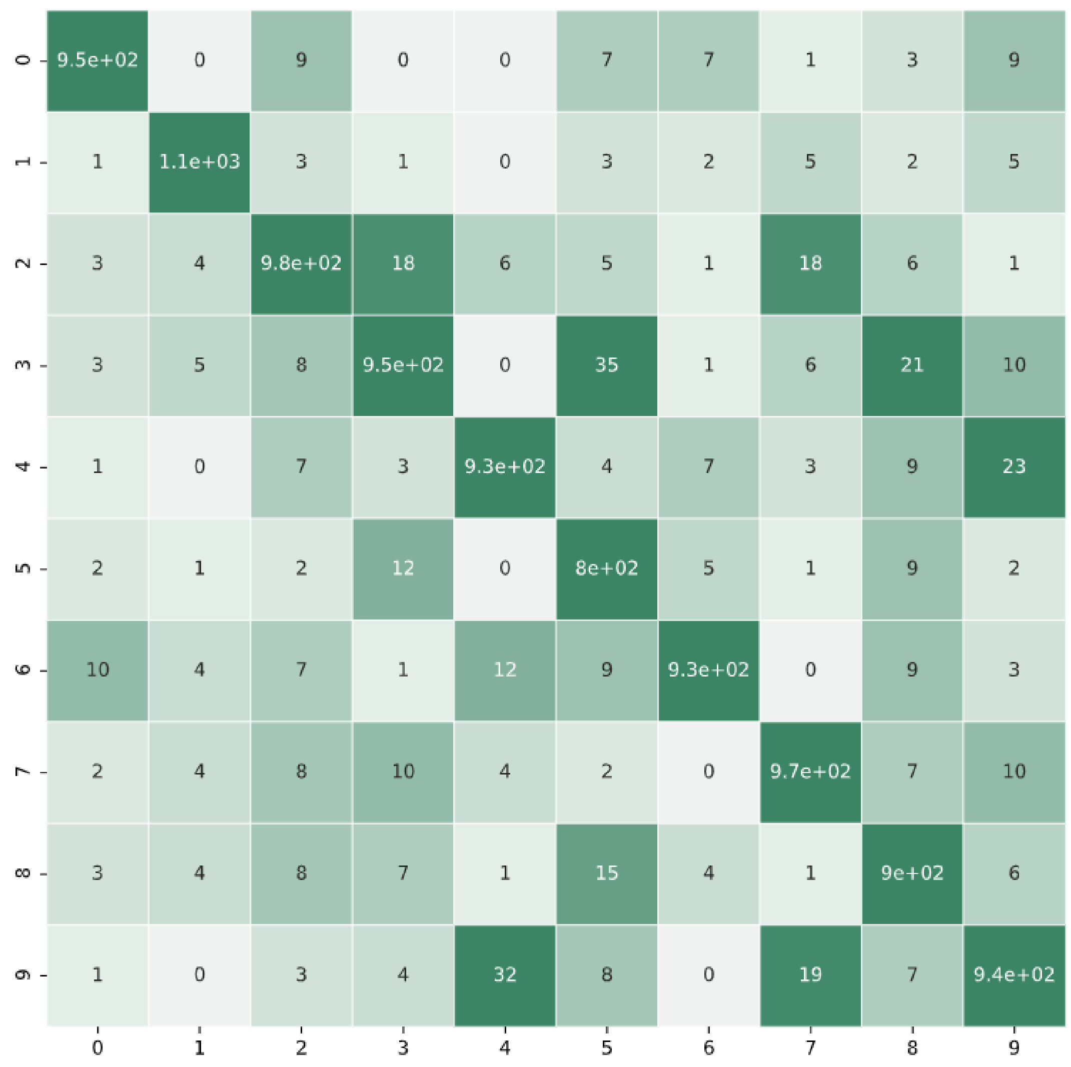
The result after testing:

- Testing accuracy: **94.75%**

Classification report:

class	precision	recall	f1-score	support
0	0.9734693878	0.9636363636	0.9685279188	990
1	0.9806167401	0.9806167401	0.9806167401	1135
2	0.9467054264	0.9403272377	0.9435055529	1039
3	0.9445544554	0.9146692234	0.9293716512	1043
4	0.9439918534	0.9420731707	0.9430315361	984
5	0.9013452915	0.9594272076	0.9294797688	838
6	0.9718162839	0.9442190669	0.95781893	986
7	0.9474708171	0.9539666993	0.9507076623	1021
8	0.9250513347	0.9484210526	0.9365904366	950
9	0.9316154609	0.9270216963	0.9293129016	1014

Confusion matrix with heatmap:



	0	1	2	3	4	5	6	7	8	9
0	9.5e+02	0	9	0	0	7	7	1	3	9
1	1	1.1e+03	3	1	0	3	2	5	2	5
2	3	4	9.8e+02	18	6	5	1	18	6	1
3	3	5	8	9.5e+02	0	35	1	6	21	10
4	1	0	7	3	9.3e+02	4	7	3	9	23
5	2	1	2	12	0	8e+02	5	1	9	2
6	10	4	7	1	12	9	9.3e+02	0	9	3
7	2	4	8	10	4	2	0	9.7e+02	7	10
8	3	4	8	7	1	15	4	1	9e+02	6
9	1	0	3	4	32	8	0	19	7	9.4e+02

Discussion

The neural network developed for this experiment gives a training accuracy of 95.814% which is decent enough. But greater accuracy can be achieved with the help of sophisticated libraries developed solely for pattern recognition and image processing.

I have used a lot of the code from M. Nielsen's GitHub repository named neural-networks-and-deep-learning. As the code available in this repository is written in Python 2.6, it is not compatible with the latest version of Python 3. So I had to change some parts of the code. Also, I have taken help from another repository maintained by Michal Daniel Dobrzanski for Python 3.

I ran the model multiple times and observed that to get the best results in accuracy it is better to take the best of 3 runs. The performance depends a lot on the learning rate. If the learning rate is too low for example 0.001, then the model performs very poorly. The same is true when the learning rate is too high such as 100.0. Hence it took some trial and errors to arrive at the final result, where the model is training with 25 epochs, mini-batch size of 64 at a learning rate of 5.0. Also, I had to update the stochastic gradient descent or SGD class to get two separate arrays, one holding the epoch numbers and the other for the percentage error at each epoch. These have been used later for plotting how the training error has reduced for each epoch while training.

After the training is complete, then testing is done. And then I used the library called sklearn to get the classification report, which then is used to generate the heatmap of confusion matrix with the help of the library called seaborn.

The model doesn't leave any stone unturned about how it implements the studied concepts like feedforward, backpropagation, mini-batch stochastic gradient descent. As the model uses computationally less expensive libraries, the model is relatively faster.

References

- neuralnetworksanddeeplearning.com
- yann.lecun.com/exdb/mnist
- github.com/mnielsen/neural-networks-and-deep-learning
- github.com/MichalDanielDobrzanski/DeepLearningPython
- scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report
- seaborn.pydata.org/generated/seaborn.heatmap