CMP304 Project Report

Handwritten Digit Recognition using Shallow Neural Network

Utkarsh Yadav 1905406

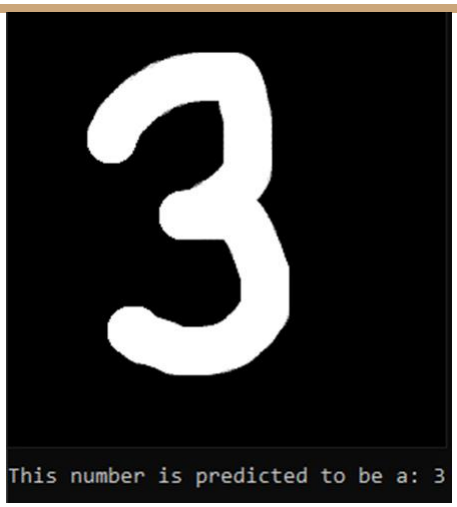

Figure 1Title Image

Introduction

A handwritten digit recognizer was developed for this project. The digit recognizer uses a shallow neural network i.e., no hidden layers, with feed-forward network architecture. The MNIST database of handwritten digits (LeChun, et al., 1998) is used to train the neural network. The MNIST database contains 60,000 images of 28x28 pixels as training data for handwritten digits ranging from 0 to 9, which converts to 784 pixels in each image, making the input layer of the neural network have 784 neurons and the output layer 10 neurons. The softmax activation function is used on the output layer as the network uses a multi-classification model and softmax forces the output to be a probability distribution which makes a lot of intuitive and mathematical sense for classification tasks (Basta, 2020). SFML graphics library is used to get user input on a 280x280 grid of pixels which is then processed and sent as a 28x28 pixel image to the neural network for evaluation and the network outputs a probability distribution from which the highest number is chosen.

User Guide

Click Left Mouse button and Drag on the window to draw a digit between 0-9

Enter to evaluate your input

C to clear the input

ESC to quit

Links

GitHub repository link: https://github.com/utkarshyadav009/Digit-Recognition-CMP304-Project

YouTube video link: https://youtu.be/QLE3xwZhBds

Background

The basic theory of the neural network and its feed-forward architecture has been taken from the Neural Networks and Deep Learning book by Michael Nielsen (Nielsen, 2019). The MNIST dataset is divided into 10 different binary files ranging from 0 to 9 and each file is loaded as a vector of floats std::vector<std::vector<float> >. The training data is loaded as a greyscale image of floats, where 0 represents white pixels and 1 represents black.

Figure 2 MNIST dataset

After the data is loaded, the input layer is initialized with an input size of 784 and output size of 10. During the initialisation, random weights are set for each of the neurons in the input and random biases are set for the output layer, these random weights, and biases range between 0 to 1 .

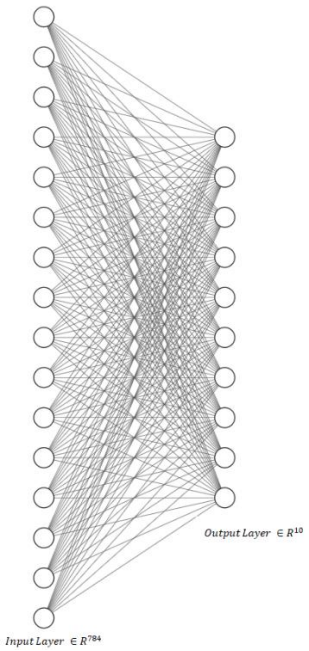

Figure 3 Neural network visual depiction

To calculate the correct weights and biases for each of the digits in the training data a cost/loss function is used.

Cost Function

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

x	Training inputs.
$y(x)$	Desired output for input ***x.***
w	Collection of all weights in the network.
b	All the biases.
n	Total number of training inputs.
a	Vector of outputs from the network where x is input.
C	Quadratic cost/loss function.

The quadratic function $C(w,b)$ is non-negative, and the cost $C(w,b)$becomes smaller i.e., $C(w,b) \approx 0$ when $y(x)$ is approximately equal to the output, a, for all training input x (Nielsen, 2019). This means if the cost function is approaching zero the prediction is closer to the desired output, on the contrary if the cost function is not approaching zero the prediction is not closer to the desired output. The goal of the training is to minimise cost as a function of weights and biases i.e., setting the weights and biases which will minimise cost as small as possible. It is achieved using the Gradient Descent algorithm.

Gradient Descent

Gradient Descent is a way to converge towards a local minimum of a cost function. A common way to visualize this is to use a visual diagram of a ball rolling down at the bottom of a valley.

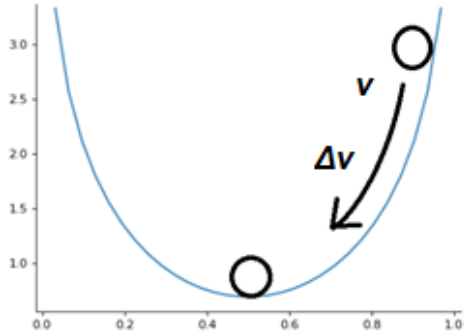

Figure 4 Gradient Descent visuals

Suppose the ball moves in the direction v and the change in distance is Δv**.** Using calculus, the change in C i.e., ΔC can be represented as follows: $\Delta C \approx \frac{\partial C}{\partial v}\Delta v$ where $\frac{\partial C}{\partial v}$ can be written as ∇C making the equation $\Delta C \approx \nabla C.\Delta v$, ∇C is called the gradient vector. This

equation is used to see how to choose Δv so that ΔC becomes negative. For example choosing Δv as $-n\nabla C$ where n is a small positive parameter also known as the learning rate gives $\Delta C \approx -n\nabla C.\nabla C = -n\|\nabla C\|^2$, as $\|\nabla C\|^2$ is always ≥ 0 the value of ΔC will always be ≤ 0. Changing the value of v with respect to $-n\nabla C$ will keep decreasing the value of C until a global minimum is achieved. The value of n should be small enough so that $\Delta C > 0$, but not too small as it would make the changes Δv miniscule.

Applying this rule to the weights and biases of the neural network will give these following equations:

$$w = w - n\frac{\partial C}{\partial w}$$

$$b = b - n\frac{\partial C}{\partial b}$$

The problem with this approach is that for each training input x a computation of the gradient ∇C_x must be performed, meaning when the number of training input increases the computation time increases as well. To counteract this stochastic gradient descent can be used to speed up learning.

Stochastic Gradient Descent

The idea behind stochastic gradient descent is randomly picking out a small number of training inputs and computing them with the gradient function, which gives a good estimation of the true gradient, hence speeding gradient descent and learning. To further elaborate, suppose a small number of training inputs are chosen randomly, let's call that m. The training inputs can be $X_1, X_2 \ldots . X_m$, this can be referred as a mini batch. It is expected that the average value of $\nabla C_{X_i}$where i is the number of elements is roughly equal to ∇C_x if the sample size of m is large enough. Putting all of that into an equation result in this, $\frac{\sum_{i=1}^{m} \nabla C_{X_i}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C$, meaning the value of ∇C can be written as $\nabla C = \frac{1}{m}\sum_{i=1}^{m} \nabla C_{X_i}$, this confirms that the overall gradient can be estimated by computing a randomly chosen mini batch.

Now applying this new equation to the weights and biases in the neural network gives these following equations:-

$$w = w - \frac{n}{m}\sum_i \frac{\partial C_{X_i}}{\partial w}$$

$$b = b - \frac{n}{m}\sum_i \frac{\partial C_{X_i}}{\partial b}$$

After training the neural network with one mini batch, another mini batch is chosen randomly for training, this is done until all the training inputs are used, this is called an epoch of training.

Softmax Activation

The softmax activation function is used as the activation function for its compatibility with multi-classification model. The softmax function turns a vector of numbers into a vector of probabilities the sum of which is always one (Basta, 2020). It outputs a vector that represents a probability distribution of a list of potential outcomes.

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

σ	Sigmoid
$\vec{z}$	Input vector
e^{z_i}	Standard exponential function for input vector
K	Number of classes in the multi-class classifier
e^{z_j}	Standard exponential function for output vector

Suppose an input vector x with size 3 is passed through the softmax activation and each element represents the hand drawn digit for the numbers $\{0, 1, 2\}$.

$x = \{2.0, 1.0, 0.1\}$ the output y for x input will be $y(x) = \{0.7, 0.2, 0.1\}$, getting the max element of that output shows that the number is predicted to be zero.

In the end the final equation for the any predicted output comes out to be this:-

$$\sigma(w_1 a_1 + w_2 a_2 + \cdots + w_n w_n + b)$$

Where w is the weight a is the activated input and b is the bias. (3Blue1Brown, 2017)

Methodology

UML Diagram

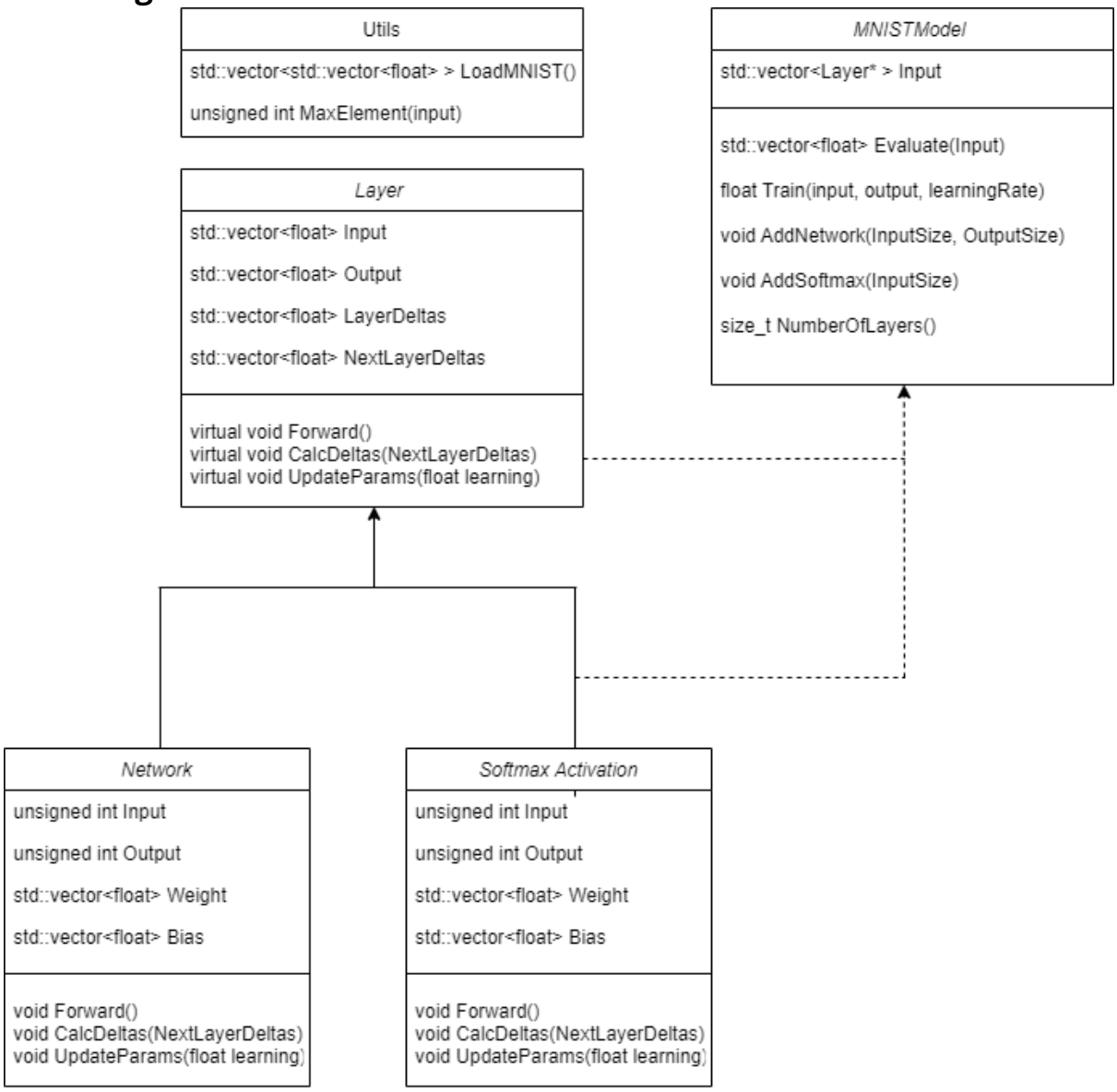

Figure 5 UML Class diagram

Explanation

The provided UML class diagram shows that both the Network and Softmax activation classes are publicly inherited from the base Layer class. The MNISTModel class is dependent on the layer as well as the network and softmax class to add both the input and output layer in the model. The Utils class is essential for loading the MNIST dataset and checking for max element in a probability distribution vector.

When the program starts the dataset is loaded into a `std::vector<std::vector<float> >` it uses std::ifstream to open the files in binary format and then loads them into a vector of floats, after the data is loaded the addNetwork and addSoftmax functions are called with 784,10 and 10 as

respective parameters. Further into the program the neural network training starts with 10 classes and 10 epochs with a learning rate of 0.1f. Using a nested for loop the model is trained with the training data and labels with variable learning rate, the formula of which is $\frac{LearningRate}{10^{\frac{epoch}{10}}}$.

In the train function the input is first evaluated against random weights and biases after which the output of that evaluation is used to calculate the deltas, weights, and biases for the network. The delta is calculated by subtracting the label vector from evaluated output vector, then the cost is calculated using the square of the delta divided by 2. Using the stochastic gradient descent function the weights and biases are calculated using a randomly selected batch of training inputs. The cost variable is then returned at the end of the train function and is displayed on the console. This model uses 10 epochs i.e., the training input is passed in 10 times to get the right delta, weights, and biases.

In the Evaluate function the data is forwarded into the input layer first, where the output is calculated by multiplying the input with the set weights, random at the start of the training, after the output is weighted, a bias is added and then the output is returned to the model where it is forwarded to the second layer for the softmax activation which returns a probability distribution of the input data.

Once the model is trained it is tested against a set of testing inputs, an accuracy is calculated for the model using the number of testing inputs and the number of correct predictions.

Once the model is trained and tested a user can input their own handwritten digits into the SFML window, to achieve this a sf vertex array of type points is used, the size of which is 280x280. After the user has inputted their digit, they can press enter to get it evaluated by the neural network, the input is then processed by averaging the RGB colours and dividing that average by 100 and normalising it, this is then pushed into a vector which is sent for evaluation. The output of the evaluation is then passed into the MaxElement function which returns the highest number in the probability distribution, the prediction is then displayed on the console.

Results

The results of the testing varied from the user inputted digits and the testing inputs. The testing input achieved an accuracy of 94.005%, on the other hand the user inputted digits accuracy averaged around 70-75% depending on how the digit is drawn.

Figure 6 Sample Digits

Above are some of the same digits drawn which were predicted right, the model seems to be very fond of the digit 5 as that was the falsest activated digit. The model doesn't seem to like the digit 9 making it hardest to predict for the model. One way to get the model to predict 9 is to draw the line after the loop a bit shorter than usual. The digits 2, 3, 4, and 5 are predicted with an average accuracy of 84%. Digits 0, 6,7 and 8 are predicted with an average accuracy of 65% and the digit nine has an accuracy of 40%. Giving the overall accuracy of the model for user inputted digits to 65%. The graphs below are for user inputted digits.

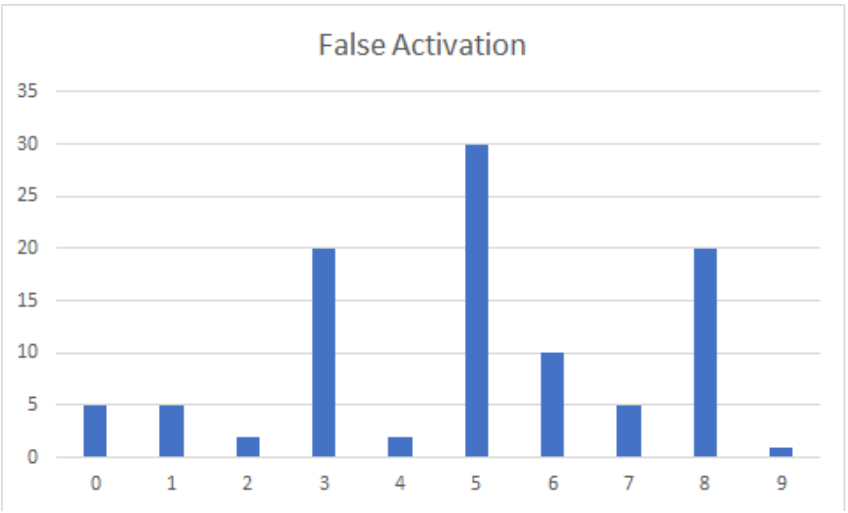

Figure 7 False Activation graph

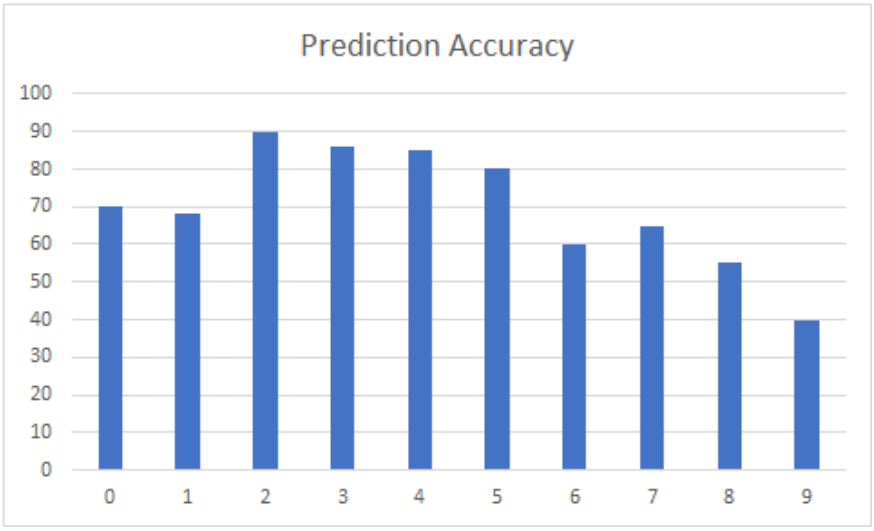

Figure 8 Prediction Accuracy Graph

Discussion

After investigating the reason for low accuracy on user inputted digits could relate to the conversion of the 280x280 grid of pixels i.e., 78,400 pixels, down to 28x28 grid of pixels i.e., 784 pixels. The conversion is done due to the training data being 28x28 pixel, but due to the conversion there is an inevitable data loss which causes the false activations. One way of improving this is using a training data which is 280x280 pixels, another way is to change the user grid down to 28x28 but after a few tries it was found that the 28x28 grid would be too small for user input. Other than that, the testing data seems to have a good enough accuracy with the softmax activation function, before implementing softmax, the sigmoid activation function was used which gave out an accuracy of 92.003% which was fine with the testing data, but the accuracy suffered a lot when trying the sigmoid with user inputted data.

Conclusion

Working on neural networks and artificial intelligence in general was quite challenging at start, due to my lack of in-depth knowledge on the subject. Most of the education material on the MNIST dataset was either written in python or used libraries like pytorch and tensorflow, so making this project in C++ was another big challenge, but after scouring the web for any relevant information or technique, I came across the Neural Networks and Deep Learning book by Michael Nielsen(*citation*) which was a great help for a beginner such as myself.

The main 3 activation function I experimented with were Sigmoid, Softmax, and ReLu. I was unable to implement ReLu properly as it kept giving a constant cost of 0.5. Sigmoid worked decently with the training data but suffered a lot with user inputted data, as I was using a multi-classification model, softmax was the best choice for this project.

On further critical evaluation maybe having a 28x28 window scaled up for user input would have been a better choice then down sampling the data. Other than that, I am happy with the progress I made and the things I learned. Another way of improving the model would have been using a wider variety of training data. To further improve the accuracy, one could implement a convolutional neural network rather than a feed-forward neural, but I do not know enough about convolutional neural networks to explain their workings.

References

3Blue1Brown, 2017. *But what is a neural network? | Chapter 1, Deep learning.* [Online]
Available at: https://www.youtube.com/watch?v=aircAruvnKk
[Accessed 22 March 2022].

Basta, N., 2020. *The Differences between Sigmoid and Softmax Activation Functions.* [Online]
Available at: https://medium.com/arteos-ai/the-differences-between-sigmoid-and-softmax-activation-function-12adee8cf322
[Accessed 27 March 2022].

Chen, S., 2016. *Handwritten Digit Recognition Using FFANN and SFML.* [Online]
Available at: https://www.youtube.com/watch?v=DB1H3Rb-6-c&t=44s
[Accessed 20 March 2022].

ContentLab, 2020. *C++ Neural Network in a Weekend.* [Online]
Available at: https://contentlab.io/c-neural-network-in-a-weekend/
[Accessed 28 March 2022].

Hansen, C., 2019. *Activation Functions Explained - GELU, SELU, ELU, ReLU and more.* [Online]
Available at: https://mlfromscratch.com/activation-functions-explained/#/
[Accessed 27 March 2022].

LeChun, Y., Cortes, C. & Burges, C. J., 1998. *The MNIST DATABASE of handwritten digits.* [Online]
Available at: http://yann.lecun.com/exdb/mnist/
[Accessed 26 March 2022].

mach, 2016. *Softmax vs Sigmoid function in Logistic classifier?.* [Online]
Available at: https://stats.stackexchange.com/questions/233658/softmax-vs-sigmoid-function-in-logistic-classifier
[Accessed 28 March 2022].

Nielsen, M., 2019. *Neural Networks and Deep Learning.* [Online]
Available at: http://neuralnetworksanddeeplearning.com/chap1.html
[Accessed 26 March 2022].

SullyChen, 2017. *Chai.* [Online]
Available at: https://github.com/SullyChen/Chai
[Accessed 26 March 2022].