Shri Vile Parle Kelvani Mandal’s

**DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING**

**Approved by AICTE and Affiliated to the University of Mumbai**

**Department of Electronics & Telecommunication Engineering**

# Mini Project On

## Title: **Handwritten Digit Recognition**

**SUBMITED BY:**

| Name of Student | SAP ID |
|---|---|
| Viren Baria | 60002160005 |
| Akshay Bhogan | 60002160007 |
| Yogesh Deshpande | 60002160021 |

Teacher’s Name: Prof. Vishakha Kelkar

Shri Vile Parle Kelvani Mandal's

**DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING**

**Approved by AICTE and Affiliated to the University of Mumbai**

**Department of Electronics & Telecommunication Engineering**

# CERTIFICATE

This is to certify that Mr. ________Viren Baria___________,

SAP ID _60002160005_ of BE EXTC 1: has submitted his/her

Mini Project for Neural Networks and Fuzzy Logic for the Academic

Year 2019-2020.

Guide Examiner

Head of Department

EXTC Department

# Index

# Introduction:

MNIST ("Modified National Institute of Standards and Technology") is the "hello world" dataset of computer vision. Since its release in 1999, this classic dataset of handwritten images has served as the basis for benchmarking classification algorithms. As new machine learning techniques emerge, MNIST remains a reliable resource for researchers and learners alike.

Handwritten digits recognition problem has been studied by researchers since 1998 with almost all the algorithms designed by then and even until now. The test error rate decreased from 12% in 1988 by linear classifier to 0.23% in 2012 by convolutional nets, and these days more and more data scientists and machine learning experts are trying to develop and validate unsupervised learning methods such as auto-encoder and deep learning model.

Digit Recognizer is a neural network formed by training a model with images of handwritten digits, using sequential neural network. The ideal output of the system is to generate the correctly recognized digit from the input handwritten digit. In this project, our goal is to correctly identify digits from a dataset of tens of thousands of handwritten images.

# Software Used: Python, Jupyter Notebook

# Flowchart:

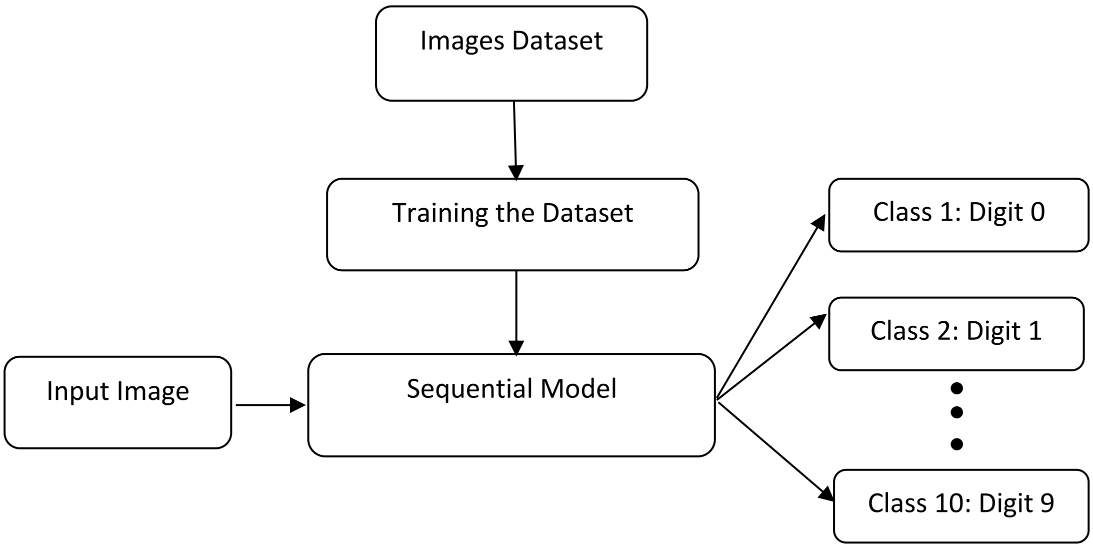



Fig.1. Digit Recognizer Flowchart

# Theory:

The dataset of Digit Recognizer is the famous **MNIST** (The dataset of handwritten digits), it can be found out at http://yann.lecun.com/exdb/mnist/ or it can be loaded in python using "mnist.load_data()", it is preloaded in keras library. This dataset consists of gray-scale images of hand-drawn digits, from zero through nine with shape 28x28 (or 784 pixels) and position in the center.

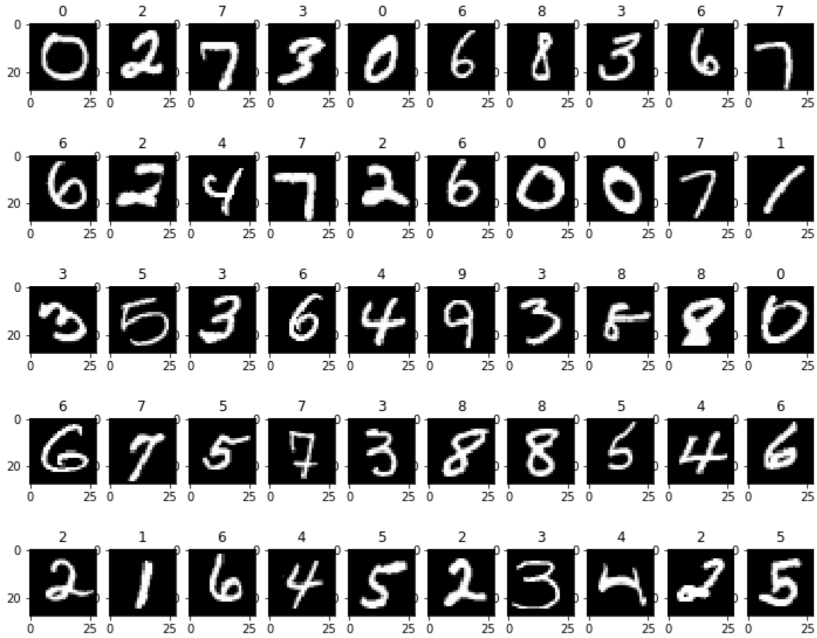


Fig.2. Input Data-Set images

Each image is 28 pixels in height and 28 pixels in width, for a total of 784 pixels in total. Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. This pixel-value is an integer between 0 and 255, inclusive.

The training data set, (train.csv), has 785 columns. The first column, called "label", is the digit that was drawn by the user. The rest of the columns contain the pixel-values of the associated image.

Each pixel column in the training set has a name like pixel_x, where x is an integer between 0 and 783, inclusive. To locate this pixel on the image, suppose that we have decomposed x as x = i * 28 + j, where i and j are integers between 0 and 27, inclusive. Then pixel_x is located on row i and column j of a 28 x 28 matrix, (indexing by zero).

NN Architecture

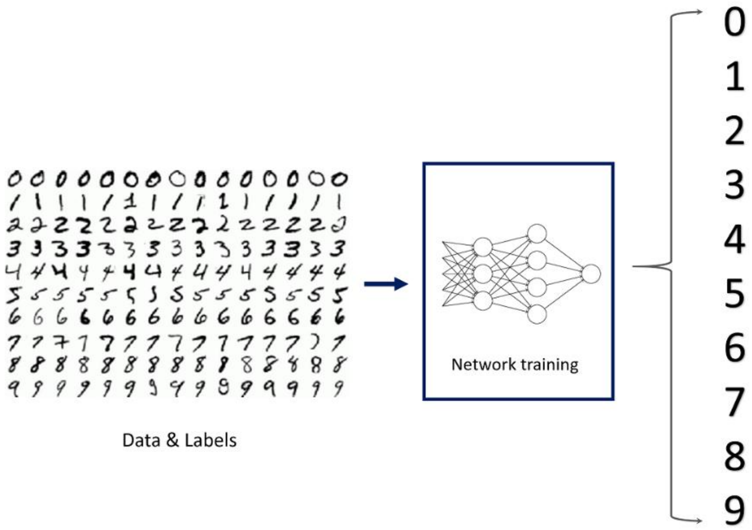



Fig.3. Generalized training process

### 1. Sigmoid or Logistic Activation Function

The Sigmoid Function curve looks like a S-shape

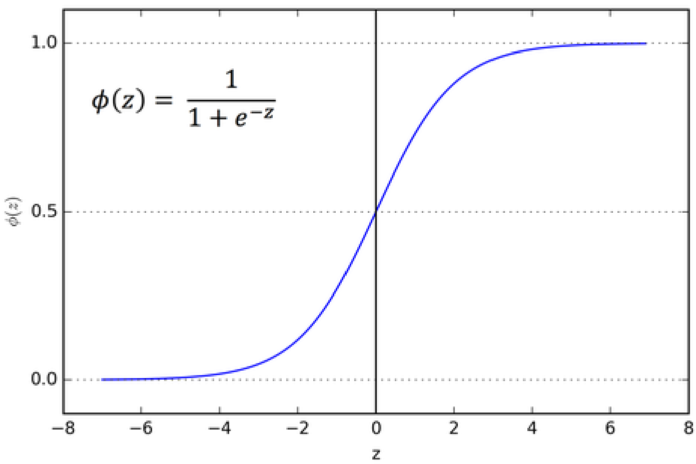



Fig.4. Sigmoid Function

The main reason why we use sigmoid function is because it exists between (0 to 1). Therefore, it is especially used for models where we have to predict the probability as an output. Since, probability of anything exists only between the range of 0 and 1, sigmoid is the right choice.

The function is differentiable. That means, we can find the slope of the sigmoid curve at any two points. The function is monotonic but function's derivative is not. The logistic sigmoid function can cause a neural network to get stuck at the training time. The softmax function is a more generalized logistic activation function which is used for multiclass classification.

### 2. ReLU (Rectified Linear Unit) Activation Function

The ReLU is the most used activation function in the world right now. Since, it is used in almost all the convolutional neural networks or deep learning.

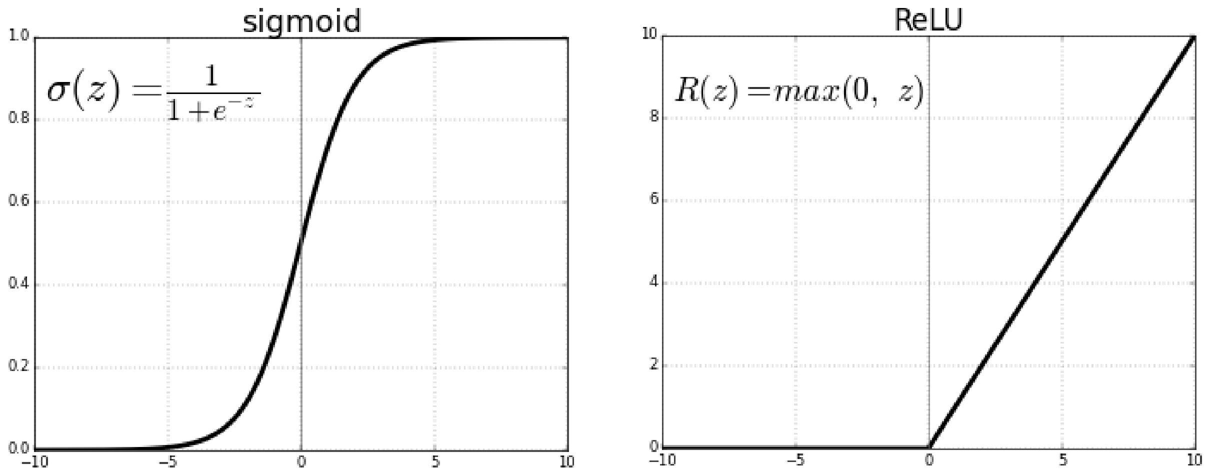



Fig.5. ReLU v/s Logistic Sigmoid

As you can see, the ReLU is half rectified (from bottom). f(z) is zero when z is less than zero and f(z) is equal to z when z is above or equal to zero.

Range: [ 0 to infinity]

The function and its derivative both are monotonic. But the issue is that all the negative values become zero immediately which decreases the ability of the model to fit or train from the data properly. That means any negative input given to the ReLU activation function turns the value into zero immediately in the graph, which in turns affects the resulting graph by not mapping the negative values appropriately.

ReLU is non-linear and has the advantage of not having any backpropagation errors unlike the sigmoid function, also for larger Neural Networks, the speed of building models based off on ReLU is very fast opposed to using Sigmoid.

### 3. Softmax Classifier

Softmax classifier provides "probabilities" for each class. Unlike the SVM which computes uncalibrated and not easy to interpret scores for all classes, the Softmax classifier allows us to compute "probabilities" for all labels.

For example, given an image the SVM classifier might give you scores [12.5, 0.6, -23.0] for the class "cat", "dog" and "ship". The softmax classifier can instead compute the probabilities of the three labels as [0.9, 0.09, 0.01], which allows you to interpret its confidence in each class. In practice, SVM and Softmax are usually comparable. The performance difference between the SVM and Softmax are usually very small, and different people will have different opinions on which classifier works better.

### 4. Adam Optimization Algorithm

Adam is different to classical stochastic gradient descent. Stochastic gradient descent maintains a single learning rate (termed alpha) for all weight updates and the learning rate does not change during training. A learning rate is maintained for each network weight (parameter) and separately adapted as learning unfolds. The method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients. Adam optimizer is combining the advantages of two other extensions of stochastic gradient descent. Specifically:

- Adaptive Gradient Algorithm (AdaGrad) that maintains a per-parameter learning rate that improves performance on problems with sparse gradients (e.g. natural language and computer vision problems).

- Root Mean Square Propagation (RMSProp) that also maintains per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight (e.g. how quickly it is changing). This means the algorithm does well on online and non-stationary problems (e.g. noisy).

### 5. Stochastic Gradient Descent (SGD)

The word stochastic means a system or a process that is linked with a random probability. Hence, in Stochastic Gradient Descent, a few samples are selected randomly instead of the whole data set for each iteration. In Gradient Descent, there is a term called "batch" which denotes the total number of samples from a dataset that is used for calculating the gradient for each iteration. In typical Gradient Descent optimization, like Batch Gradient Descent, the batch is taken to be the whole dataset. Although, using the whole dataset is really useful for getting to the minima in a less noisy or less random manner, but the problem arises when our datasets get really huge.

Suppose, you have a million samples in your dataset, so if you use a typical Gradient Descent optimization technique, you will have to use all of the one million samples for completing one iteration while performing the Gradient Descent, and it has to be done for every iteration until the minima is reached. Hence, it becomes computationally very expensive to perform.

This problem is solved by Stochastic Gradient Descent. In SGD, it uses only a single sample, i.e., a batch size of one, to perform each iteration. The sample is randomly shuffled and selected for performing the iteration.

# Results:

**Model 1: Single MLP**

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 10)                7850
=================================================================
Total params: 7,850
Trainable params: 7,850
Non-trainable params: 0
```

**Model 2: MLP + Sigmoid + SGD Optimizer**

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 512)               401920
_________________________________________________________________
dense_3 (Dense)              (None, 128)               65664
_________________________________________________________________
dense_4 (Dense)              (None, 10)                1290
=================================================================
Total params: 468,874
Trainable params: 468,874
Non-trainable params: 0
```

**Model 3: MLP + Sigmoid + ADAM Optimizer**

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_5 (Dense)              (None, 512)               401920
_________________________________________________________________
dense_6 (Dense)              (None, 128)               65664
_________________________________________________________________
dense_7 (Dense)              (None, 10)                1290
=================================================================
Total params: 468,874
Trainable params: 468,874
Non-trainable params: 0
```

**Model 4: MLP + RELU + SGD Optimizer**

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_8 (Dense)              (None, 512)               401920
_________________________________________________________________
dense_9 (Dense)              (None, 128)               65664
_________________________________________________________________
dense_10 (Dense)             (None, 10)                1290
=================================================================
Total params: 468,874
Trainable params: 468,874
Non-trainable params: 0
```

**Model 5: MLP + RELU +ADAM**

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_11 (Dense)             (None, 512)               401920
_________________________________________________________________
dense_12 (Dense)             (None, 128)               65664
_________________________________________________________________
dense_13 (Dense)             (None, 10)                1290
=================================================================
Total params: 468,874
Trainable params: 468,874
Non-trainable params: 0
```

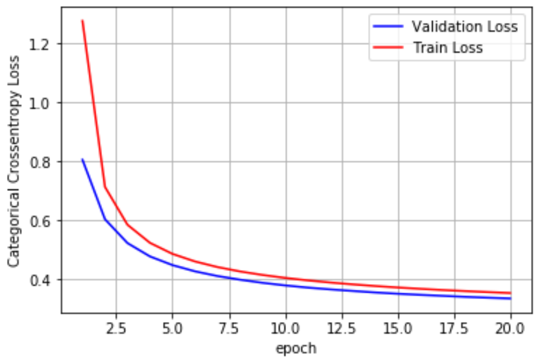
**Model 6: Batch Normalization (MLP + Sigmoid + SoftMax + ADAM Optimizer)**

```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_14 (Dense)             (None, 512)               401920
_________________________________________________________________
batch_normalization_1 (Batch (None, 512)               2048
_________________________________________________________________
dense_15 (Dense)             (None, 128)               65664
_________________________________________________________________
batch_normalization_2 (Batch (None, 128)               512
_________________________________________________________________
dense_16 (Dense)             (None, 10)                1290
=================================================================
Total params: 471,434
Trainable params: 470,154
Non-trainable params: 1,280
```

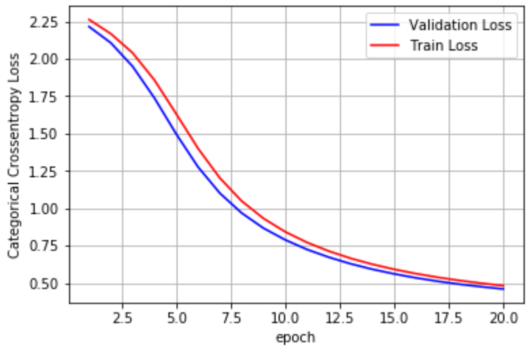
**Model 7: MLP + DROPOUT + ADAM OPTIMIZER**

```
Model: "sequential_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_17 (Dense)             (None, 512)               401920
_________________________________________________________________
batch_normalization_3 (Batch (None, 512)               2048
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_18 (Dense)             (None, 128)               65664
_________________________________________________________________
batch_normalization_4 (Batch (None, 128)               512
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_19 (Dense)             (None, 10)                1290
=================================================================
Total params: 471,434
Trainable params: 470,154
Non-trainable params: 1,280
```

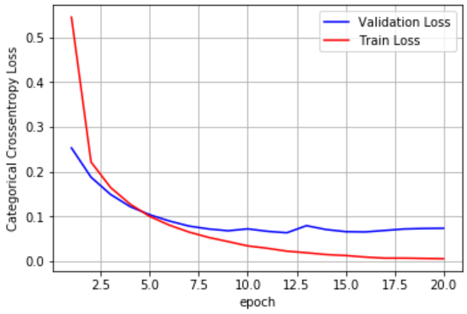
| Model used | Test Accuracy |
|---|---|
| Model 1 | 0.909500002861023 |
| Model 2 | 0.8809999823570251 |
| Model 3 | 0.98089998960495 |
| Model 4 | 0.9649999737739563 |
| Model 5 | 0.9804999828338623 |
| Model 6 | 0.9745000004768372 |
| Model 7 | 0.9692000150680542 |



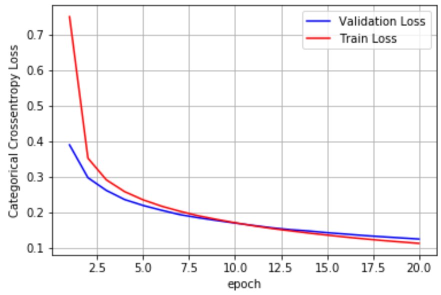

Model 1



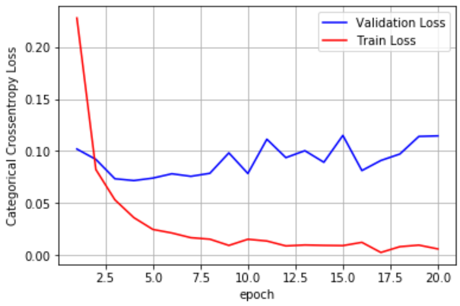

Model 2



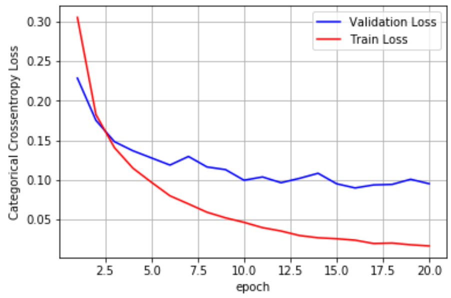

Model 3



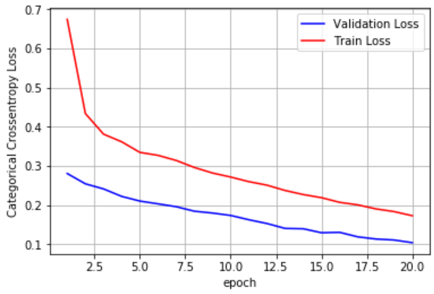

Model 4




Model 5




Model 6




Model 7

# Conclusion:

We have successfully performed the training as well as testing of the different models using handwritten digit images. The graphs above give the plots of training loss and validation loss. The accuracy is used as a measure to check the performance of the neural network. The highest accuracy is obtained in Model 3: MLP + Sigmoid + ADAM Optimizer, which has the accuracy of 0.98089998960495. The output of the system are predicted and stored as y_labels.