CHAPTER 1

INTRODUCTION

1.1 Introduction

In this chapter we will discuss the area of our work, how is the present-day scenario with regard to the work area, what is our motivation to do this project, significance of the possible end result, objective of our work, main work objective, importance of the end result and organization of project report.

The project is to build a deep learning model which can classify hand written assamese digits. Handwritten digit recognition is the ability of a computer to recognize the human handwritten digits from different sources like images, papers, touch screens, etc, and classify them into 10 predefined classes(0-9). This has been a topic of boundless-research in the field of deep learning. With the rise of deep learning, different neural networks are introduced to generate text. For example, researchers use the convolutional neural network (CNN), artificial neural network (ANN) because they are very effective for image recognition and identifying different image patterns.

1.2 Motivation

While the field of handwritten digit classification remains largely unexplored, there are already quite a few areas where we find its applications. Most popular use cases of digit classification have emerged in many fields like number plate recognition, postal mail sorting, bank check processing, etc. Handwritten digit or character recognition is one of the practically important issues in pattern recognition applications. But it has been observed that a particular amount of these digits are being targeted for the optical character recognition in Assamese languages. In such kind of handwritten digit recognition and classification we face various kinds of challenges due to the reason that different people have different types of handwriting such that it is not an optical character recognition. Despite the fact that Assamese language is widely spoken across the North-East India. This has motivated us to device a computational model for the automatic recognition of the handwritten Assamese digits belonging to 10 different classes.

Our objective is to build a deep learning model that can classify Assamese handwritten digits. To simply say model this will detect handwritten Assamese digits from the image provided.

CHAPTER 2

LITERATURE REVIEW

This chapter will include discussion on the project title, present day scenario or recent development in the field and also the background history of natural language generation.

2.1 Project Title

Our main objective is to build a deep learning model that can classify handwritten assamese digits.

2.2 Literature Review

There has very good developments in this field recently. The most popular techniques for the prediction and classification of handwritten digit classification using convolutional neural network (CNN) and Artificial Neural Network:

- DigiNet: The paper by Prarthana Dutta, Naresh Babu Muppalaneni, employs a convolutional neural network model to learn and understand the various styles of handwritten digits. This model exercises the convolutional neural network composed of six alternative convolution and pooling layers and is able to attain state-of-the-art performance of the Assamese handwritten digits.

- MNIST Handwritten Digits Classification using a Convolutional Neural Network (CNN): This paper by Krut Patel proposes a simple neural network approach towards handwritten digit recognition using convolution. Here convolution neural networks are implemented with an MNIST dataset of 70000 digits with 250 distinct forms of writings.

Background theory:

A Convolutional Neural Network (ConvNet/CNN) is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNets have the ability to learn these filters/characteristics.

The architecture of a ConvNet is analogous to that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex. Individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlap to cover the entire visual area.

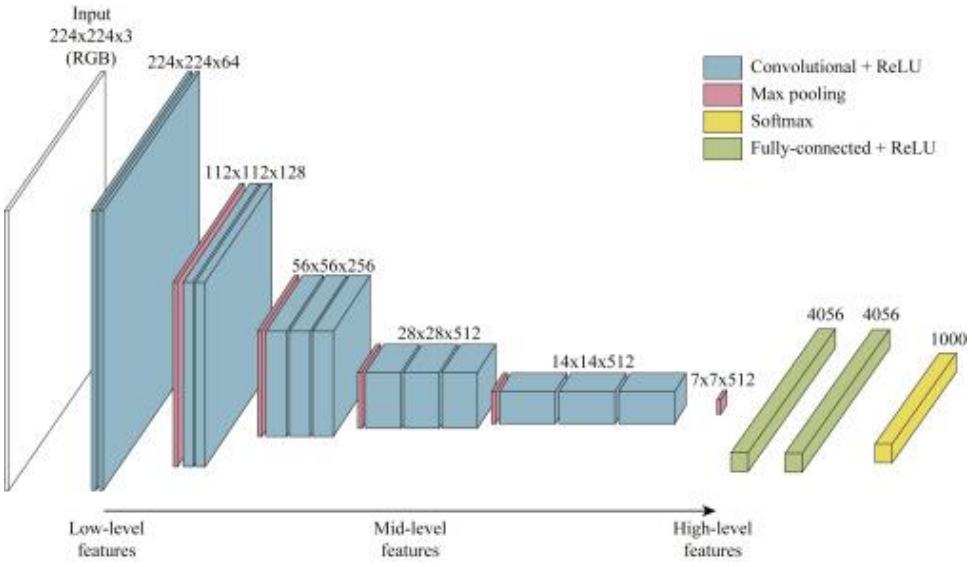

Fig 2.1 : Convolutional Neural Network

Artificial neural network (ANN) model involves computations and mathematics, which simulate the human–brain processes. Many of the recently achieved advancements are related to the artificial intelligence research area such as image and voice recognition, robotics, and using ANNs. The ANN models have the specific architecture format, which is inspired by a biological nervous system. Like the structure of the human brain, the ANN models consist of neurons in a complex and nonlinear form. The neurons are connected to each other by weighted links. All the processes in ANN models, such as data collection and analysis, network structure design, number of hidden layers, network simulation, and weights/bias trade-off, are computed through learning and training methods. The ANN applications solving problems of the real world include a wide range of scientific fields from finance to hydrology and fall into the three categories: (i) pattern classification, (ii) prediction, and (iii) control and optimization.

ANNs consist of a layer of input nodes and layer of output nodes, connected by one or more layers of hidden nodes. Input layer nodes pass information to hidden layer nodes by firing activation functions, and hidden layer nodes fire or remain dormant depending on the evidence presented. The hidden layers apply weighting functions to the evidence, and when the value of a particular node or set of nodes in the hidden layer reaches some threshold, a value is passed to one or more nodes in the output layer.

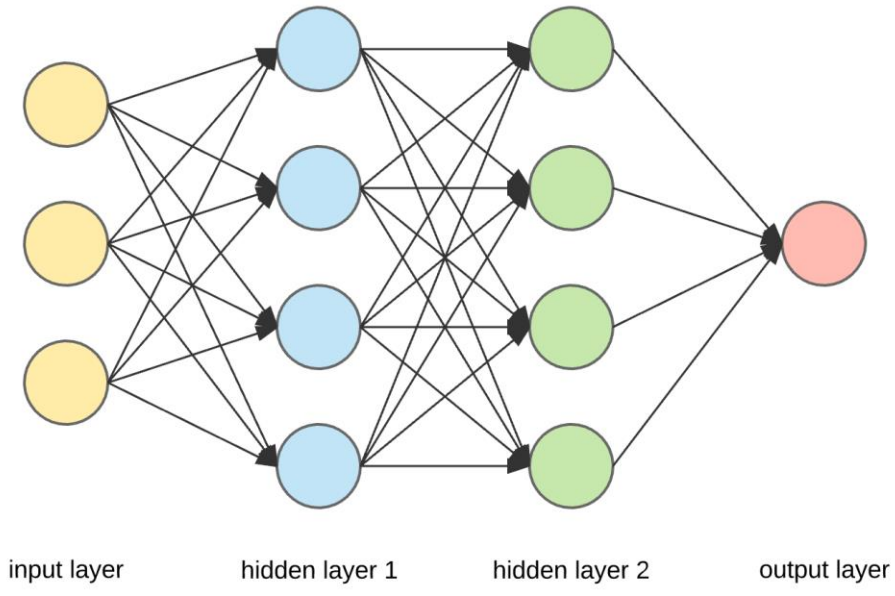

Fig 2.2: Artificial Neural Network

CHAPTER 3

METHODOLOGY

In this chapter we will be discussing the detailed methodology of the project, algorithms

3.1Methodology

Here is the step-by-step procedure that we followed

3.2 Dataset

First of all, we needed to select the dataset and we took Assamese Digit Dataset form IEEE dataport. This dataset comes up as a benchmark dataset for machines to automatically recognizing the handwritten assamese digists (numerals) by extracting useful features by analyzing the structure. The Assamese language comprises of a total of 10 digits from 0 to 9. It contains a total of 516 images, among them 387 images (75%) are taken in the training set and 129 images (25%) in the testing set.

3.3 Data pre-processing

Images come in different shapes and sizes**.** They also come through different sources. Taking all these variations into consideration, we need to perform some pre-processing on any image data. Also, among the first step of data pre-processing is to make the images of the same size**.** One of the simpler operations where we take all the pixels whose intensities are above a certain threshold and convert them to ones; the pixels having value less than the threshold are converted to zero. This results in a binary image.

All the images were standardized by subtracting the mean activity over the training set from each pixel and dividing by their standard deviation. As we know, every image consists of pixel values between 0 and 255.Normalisation is the most crucial step in the pre-processing part. This refers to rescaling the pixel values so that they lie within a confined range. One of the reasons to do this is to help with the issue of propagating gradients. So we divide every value by 255 to scale the data from [0..255] to [0..1]. It helps the model to better learning of features by decreasing computational complexities if we have data that scales bigger.

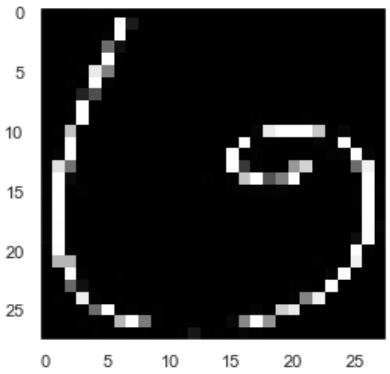

Fig 3.1: Original Image

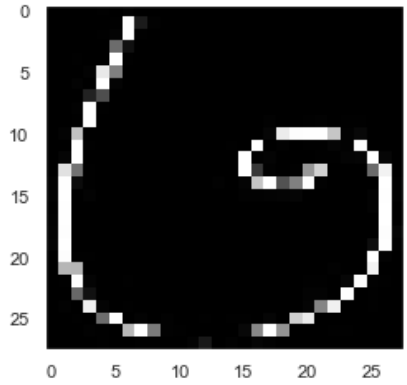

Fig 3.2 : Image with normalized pixel values

3.4 Training and Validation split

We are segmenting the input data for training into two exclusive data namely, Train and Validation data sets. Train data is used to train the model whereas the validation data is used for cross verification of the model's accuracy and how well the model is generalized for the data other than training data. Validation accuracy and loss will tell us the performance of the model for new data and it will show if there is overfitting or underfitting situation while model training.

3.5 Define the neural network

We have used CNN, ANN neural network. Building models in Python alone is hard so using framework make it easy. These days there are different popular framework like Pytorch, Keras, TensorFlow exist. We choose Keras because it is easier to use. Keras API uses TensorFlow in the backend.

```
In [8]: model.summary()
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 32)        832
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 28, 28, 32)        25632
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 32)        0
_________________________________________________________________
dropout (Dropout)            (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 14, 64)        18496
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 14, 14, 64)        36928
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
dropout_1 (Dropout)          (None, 7, 7, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 3136)              0
_________________________________________________________________
dense (Dense)                (None, 256)               803072
_________________________________________________________________
dropout_2 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                2570
=================================================================
Total params: 887,530
Trainable params: 887,530
Non-trainable params: 0
```

Fig 3.3 : CNN Architecture Used

```
In [12]: classifier.summary()
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 784)               615440
_________________________________________________________________
dense_1 (Dense)              (None, 128)               100480
_________________________________________________________________
dense_2 (Dense)              (None, 128)               16512
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense_3 (Dense)              (None, 64)                8256
_________________________________________________________________
dense_4 (Dense)              (None, 64)                4160
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_5 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_6 (Dense)              (None, 32)                1056
_________________________________________________________________
dropout_2 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_7 (Dense)              (None, 10)                330
=================================================================
Total params: 748,314
Trainable params: 748,314
Non-trainable params: 0
```

Fig 3.4: ANN architecture used

3.6 Training

Models have different configurable variables that change its capability, we trained models with different configurations and kept the best performing model.

- For training the model is provided with an input shape of (28, 28, 1), batch size of 32, categorical crossentropy as the loss function and 0.001 as the learning rate. The mentioned parameters are taken same for both the architectures.
- Training is done with different epoch value, where epoch is number of times entire data is iterated during training.

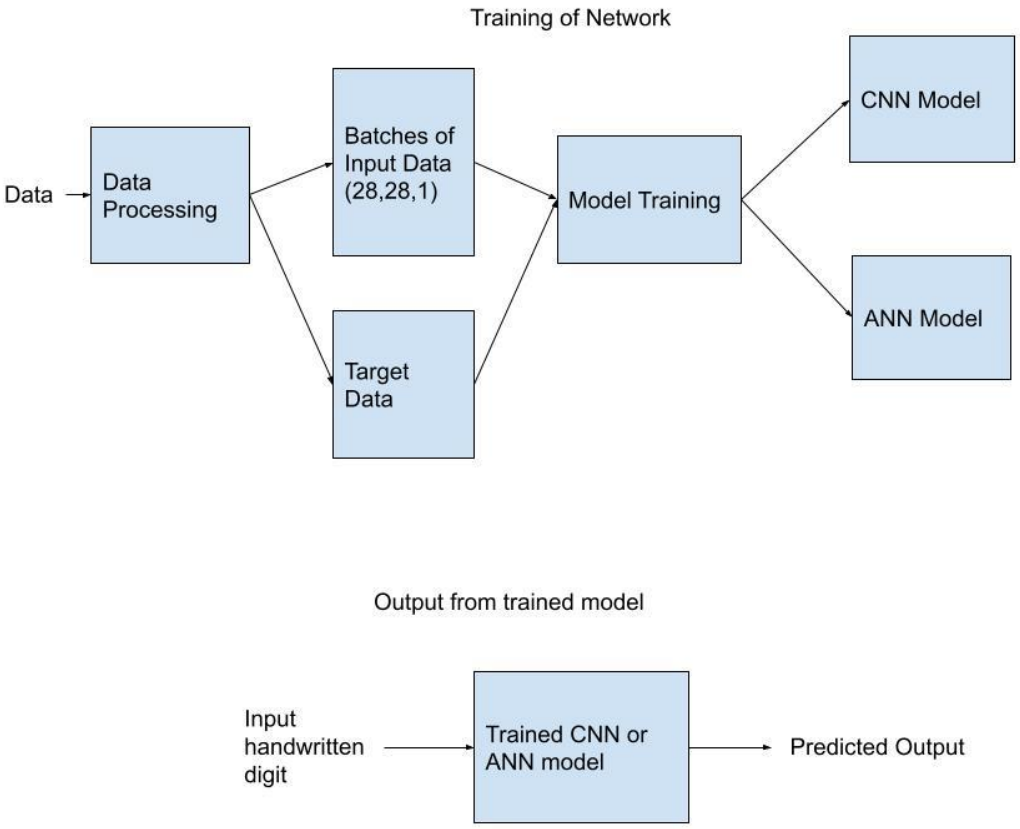

Fig 3.5: Functional block diagram

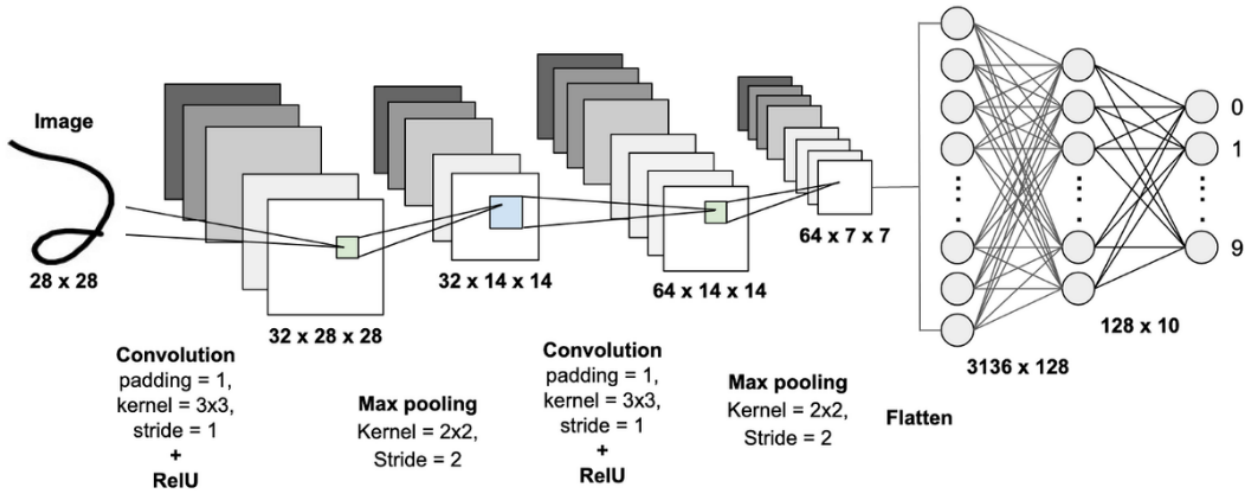

Fig 3.6: Block diagram showing working of CNN

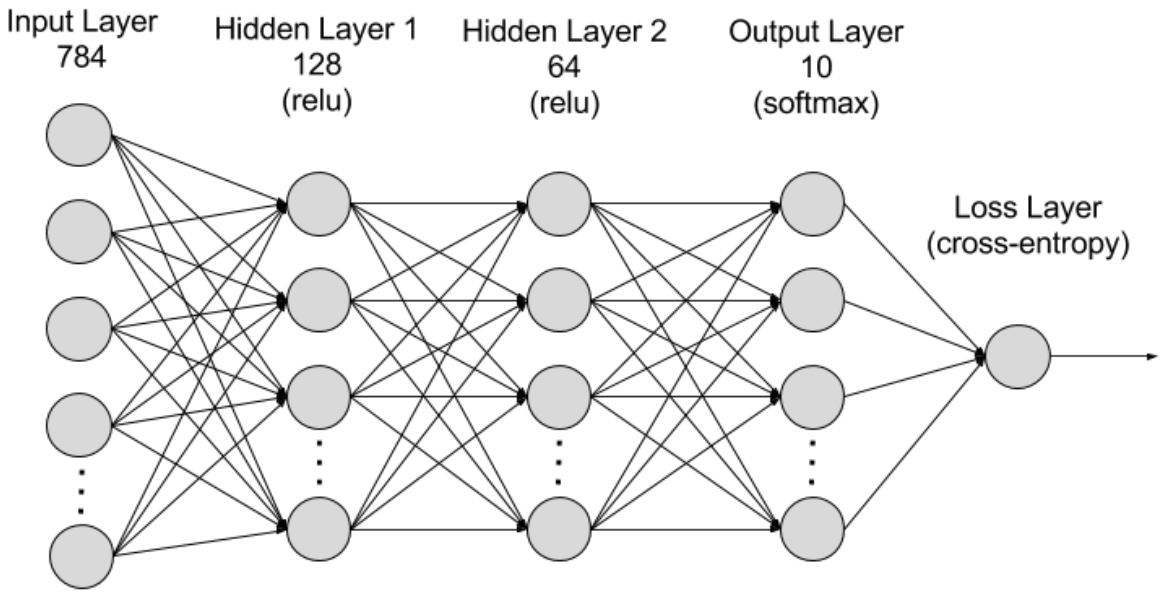

Fig 3.7: Block diagram showing working of ANN

Tools used

Programming language:

- Python: It is an easy-to-use programming language, it has a rich eco system of machine learning libraries which are used to build and train machine learning model easily.

Framework:

- Keras: Keras is an open-source software library that provides a Python interface for artificial neural networks. Keras acts as an interface for the TensorFlow library.
- Tensorflow: Tensorflow is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in ML

CHAPTER 4

RESULT ANALYSIS

In this chapter we will be discussing the detailed results obtained after training the CNN and ANN models.

4.1 Result Analysis

Here we will discuss the obtained results separately for both the models.

4.2 CNN model

The CNN model has trained up to 40 epochs, for which the batch size is 32.The train loss value began with 2.4159 and validation loss with 2.2983

```
Epoch 1/40
14/14 - 4s - loss: 2.4159 - accuracy: 0.0898 - val_loss: 2.2983 - val_accuracy: 0.0980
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_acc` which is not available. Available metrics are:
loss,accuracy,val_loss,val_accuracy,lr
Epoch 2/40
14/14 - 2s - loss: 2.3041 - accuracy: 0.1040 - val_loss: 2.2821 - val_accuracy: 0.0980
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_acc` which is not available. Available metrics are:
loss,accuracy,val_loss,val_accuracy,lr
Epoch 3/40
14/14 - 2s - loss: 2.2624 - accuracy: 0.1466 - val_loss: 2.1558 - val_accuracy: 0.3529
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_acc` which is not available. Available metrics are:
loss,accuracy,val_loss,val_accuracy,lr
Epoch 4/40
14/14 - 2s - loss: 1.8957 - accuracy: 0.3712 - val_loss: 1.9791 - val_accuracy: 0.3725
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_acc` which is not available. Available metrics are:
loss,accuracy,val_loss,val_accuracy,lr
Epoch 5/40
14/14 - 2s - loss: 1.4089 - accuracy: 0.5508 - val_loss: 0.8451 - val_accuracy: 0.7059
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_acc` which is not available. Available metrics are:
loss,accuracy,val_loss,val_accuracy,lr
Epoch 6/40
14/14 - 2s - loss: 1.1154 - accuracy: 0.6265 - val_loss: 0.7081 - val_accuracy: 0.6863
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_acc` which is not available. Available metrics are:
loss,accuracy,val_loss,val_accuracy,lr
Epoch 7/40
14/14 - 2s - loss: 0.9107 - accuracy: 0.6856 - val_loss: 0.5164 - val_accuracy: 0.8235
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_acc` which is not available. Available metrics are:
loss,accuracy,val_loss,val_accuracy,lr
Epoch 8/40
14/14 - 2s - loss: 0.8995 - accuracy: 0.6832 - val_loss: 0.5565 - val_accuracy: 0.7843
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_acc` which is not available. Available metrics are:
loss,accuracy,val loss,val accuracy,lr
```

Fig 4.1: Loss and Accuracy at the beginning of training CNN

At 40 epoch, the train accuracy is 92.67% and validation accuracy is 92.16% . The train loss went up to 0.2419 and validation loss to 0.2851.

```
14/14 - 2s - loss: 0.2527 - accuracy: 0.9267 - val_loss: 0.3350 - val_accuracy: 0.8824
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_acc` which is not available. Available metrics are:
loss,accuracy,val_loss,val_accuracy,lr
Epoch 32/40
14/14 - 2s - loss: 0.3046 - accuracy: 0.8723 - val_loss: 0.2943 - val_accuracy: 0.9216
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_acc` which is not available. Available metrics are:
loss,accuracy,val_loss,val_accuracy,lr
Epoch 33/40
14/14 - 2s - loss: 0.2592 - accuracy: 0.9125 - val_loss: 0.2779 - val_accuracy: 0.9020
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_acc` which is not available. Available metrics are:
loss,accuracy,val_loss,val_accuracy,lr
Epoch 34/40
14/14 - 3s - loss: 0.2325 - accuracy: 0.9173 - val_loss: 0.3558 - val_accuracy: 0.8824
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_acc` which is not available. Available metrics are:
loss,accuracy,val_loss,val_accuracy,lr
Epoch 35/40
14/14 - 2s - loss: 0.1630 - accuracy: 0.9362 - val_loss: 0.3526 - val_accuracy: 0.8824
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_acc` which is not available. Available metrics are:
loss,accuracy,val_loss,val_accuracy,lr
Epoch 36/40
14/14 - 2s - loss: 0.1940 - accuracy: 0.9267 - val_loss: 0.3924 - val_accuracy: 0.8824
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_acc` which is not available. Available metrics are:
loss,accuracy,val_loss,val_accuracy,lr
Epoch 37/40
14/14 - 2s - loss: 0.1772 - accuracy: 0.9385 - val_loss: 0.3685 - val_accuracy: 0.9216
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_acc` which is not available. Available metrics are:
loss,accuracy,val_loss,val_accuracy,lr
Epoch 38/40
14/14 - 2s - loss: 0.2002 - accuracy: 0.9385 - val_loss: 0.4515 - val_accuracy: 0.9020
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_acc` which is not available. Available metrics are:
loss,accuracy,val_loss,val_accuracy,lr
Epoch 39/40
14/14 - 2s - loss: 0.2000 - accuracy: 0.9267 - val_loss: 0.3986 - val_accuracy: 0.9216
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_acc` which is not available. Available metrics are:
loss,accuracy,val_loss,val_accuracy,lr
Epoch 40/40
14/14 - 2s - loss: 0.2419 - accuracy: 0.9267 - val_loss: 0.2851 - val_accuracy: 0.9216
WARNING:tensorflow:Learning rate reduction is conditioned on metric `val_acc` which is not available. Available metrics are:
loss,accuracy,val_loss,val_accuracy,lr
```

Fig 4.2: Loss and Accuracy at the end of training

Plot for Training Loss Vs Validation Loss and Training Accuracy Vs Validation Accuracy

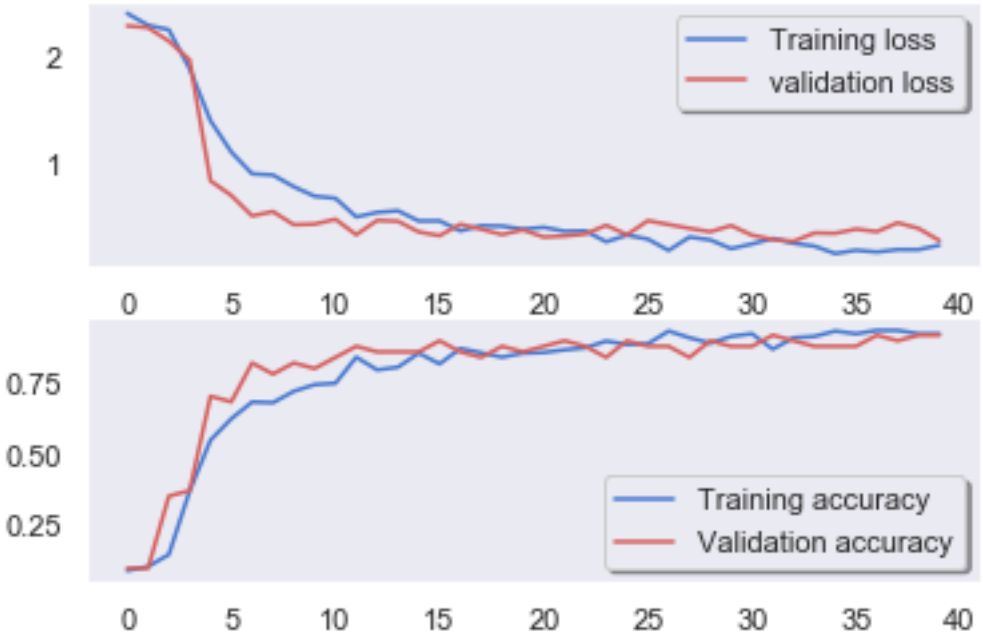

Fig 4.3: Train loss Vs Valid loss and Train Accuracy Vs Valid Accuracy for CNN model training

The confusion matrix plot for CNN is below,

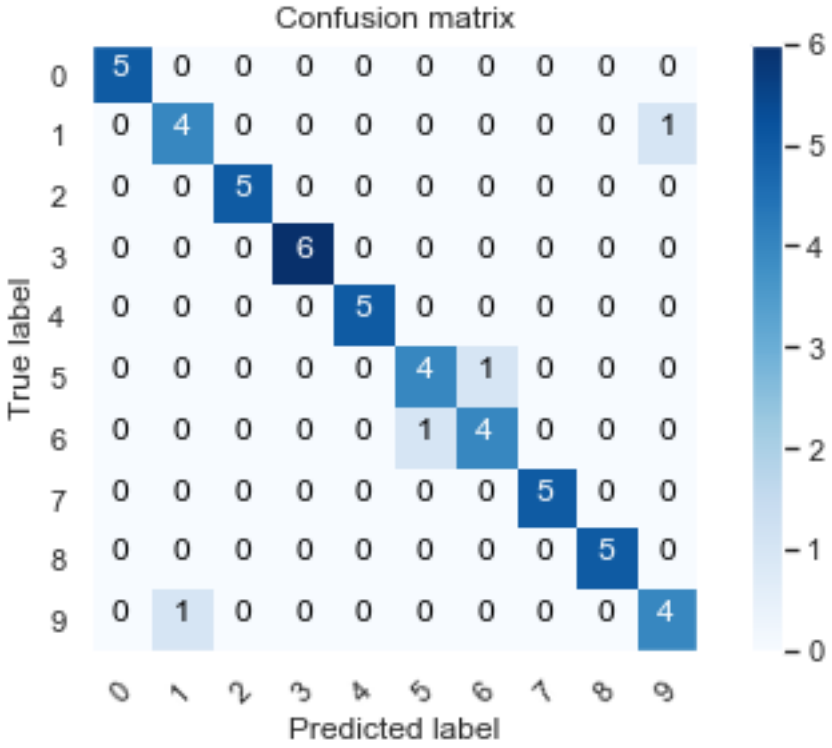

Fig 4.4: Confusion matrix plot for CNN model

4.3 ANN model

The ANN model has trained upto 40 epochs, for which the batch size is 32.The train loss value began with 2.3083 and valid loss began with 2.3043

```
Epoch 1/40
15/15 [==============================] - 1s 20ms/step - loss: 2.3083 - accuracy: 0.1099 - val_loss: 2.3043 - val_accuracy: 0.0980
Epoch 2/40
15/15 [==============================] - 0s 7ms/step - loss: 2.3071 - accuracy: 0.0967 - val_loss: 2.2985 - val_accuracy: 0.0980
Epoch 3/40
15/15 [==============================] - 0s 7ms/step - loss: 2.3042 - accuracy: 0.1121 - val_loss: 2.2980 - val_accuracy: 0.0980
Epoch 4/40
15/15 [==============================] - 0s 7ms/step - loss: 2.2981 - accuracy: 0.1033 - val_loss: 2.2967 - val_accuracy: 0.0980
Epoch 5/40
15/15 [==============================] - 0s 7ms/step - loss: 2.3029 - accuracy: 0.1165 - val_loss: 2.2936 - val_accuracy: 0.1373
Epoch 6/40
15/15 [==============================] - 0s 6ms/step - loss: 2.2972 - accuracy: 0.1121 - val_loss: 2.2943 - val_accuracy: 0.0980
Epoch 7/40
15/15 [==============================] - 0s 5ms/step - loss: 2.2962 - accuracy: 0.1121 - val_loss: 2.2971 - val_accuracy: 0.0588
Epoch 8/40
15/15 [==============================] - 0s 5ms/step - loss: 2.2915 - accuracy: 0.1363 - val_loss: 2.2887 - val_accuracy: 0.1373
Epoch 9/40
15/15 [==============================] - 0s 5ms/step - loss: 2.2968 - accuracy: 0.1099 - val_loss: 2.2925 - val_accuracy: 0.1961
Epoch 10/40
15/15 [==============================] - 0s 5ms/step - loss: 2.2921 - accuracy: 0.1253 - val_loss: 2.2935 - val_accuracy: 0.2353
Epoch 11/40
15/15 [==============================] - 0s 5ms/step - loss: 2.2829 - accuracy: 0.1538 - val_loss: 2.2840 - val_accuracy: 0.1569
Epoch 12/40
15/15 [==============================] - 0s 5ms/step - loss: 2.2825 - accuracy: 0.1473 - val_loss: 2.2837 - val_accuracy: 0.1373
Epoch 13/40
15/15 [==============================] - 0s 6ms/step - loss: 2.2824 - accuracy: 0.1341 - val_loss: 2.2786 - val_accuracy: 0.1569
Epoch 14/40
15/15 [==============================] - 0s 5ms/step - loss: 2.2777 - accuracy: 0.1538 - val_loss: 2.2757 - val_accuracy: 0.1765
Epoch 15/40
15/15 [==============================] - 0s 7ms/step - loss: 2.2742 - accuracy: 0.1473 - val_loss: 2.2759 - val_accuracy: 0.1765
Epoch 16/40
15/15 [==============================] - 0s 9ms/step - loss: 2.2747 - accuracy: 0.1516 - val_loss: 2.2635 - val_accuracy: 0.1961
```

Fig 4.5 : Loss and Accuracy at the beginning of training ANN

At 40 epoch, the train accuracy is 24.48% and validation accuracy is 22.22% . The train loss went up to 2.0602 and validation loss to 2.2229.

```
15/15 [==============================] - 0s 8ms/step - loss: 2.2229 - accuracy: 0.1956 - val_loss: 2.2002 - val_accuracy: 0.2353
Epoch 24/40
15/15 [==============================] - 0s 9ms/step - loss: 2.2083 - accuracy: 0.1890 - val_loss: 2.3235 - val_accuracy: 0.1176
Epoch 25/40
15/15 [==============================] - 0s 8ms/step - loss: 2.2156 - accuracy: 0.1868 - val_loss: 2.2577 - val_accuracy: 0.0980
Epoch 26/40
15/15 [==============================] - 0s 8ms/step - loss: 2.1813 - accuracy: 0.1934 - val_loss: 2.1879 - val_accuracy: 0.1961
Epoch 27/40
15/15 [==============================] - 0s 8ms/step - loss: 2.2066 - accuracy: 0.2022 - val_loss: 2.2769 - val_accuracy: 0.1373
Epoch 28/40
15/15 [==============================] - 0s 8ms/step - loss: 2.1905 - accuracy: 0.2132 - val_loss: 2.1541 - val_accuracy: 0.2353
Epoch 29/40
15/15 [==============================] - 0s 8ms/step - loss: 2.1437 - accuracy: 0.2286 - val_loss: 2.2074 - val_accuracy: 0.1176
Epoch 30/40
15/15 [==============================] - 0s 8ms/step - loss: 2.1473 - accuracy: 0.2154 - val_loss: 2.3428 - val_accuracy: 0.1569
Epoch 31/40
15/15 [==============================] - 0s 8ms/step - loss: 2.1723 - accuracy: 0.2352 - val_loss: 2.0858 - val_accuracy: 0.2745
Epoch 32/40
15/15 [==============================] - 0s 9ms/step - loss: 2.1159 - accuracy: 0.2440 - val_loss: 2.2014 - val_accuracy: 0.1569
Epoch 33/40
15/15 [==============================] - 0s 8ms/step - loss: 2.1257 - accuracy: 0.2242 - val_loss: 2.0583 - val_accuracy: 0.3922
Epoch 34/40
15/15 [==============================] - 0s 9ms/step - loss: 2.1172 - accuracy: 0.2593 - val_loss: 2.3065 - val_accuracy: 0.1176
Epoch 35/40
15/15 [==============================] - 0s 8ms/step - loss: 2.0628 - accuracy: 0.2440 - val_loss: 2.4340 - val_accuracy: 0.0980
Epoch 36/40
15/15 [==============================] - 0s 8ms/step - loss: 2.1122 - accuracy: 0.2308 - val_loss: 2.1660 - val_accuracy: 0.2941
Epoch 37/40
15/15 [==============================] - 0s 9ms/step - loss: 2.0437 - accuracy: 0.2571 - val_loss: 2.0807 - val_accuracy: 0.2353
Epoch 38/40
15/15 [==============================] - 0s 8ms/step - loss: 2.0928 - accuracy: 0.2418 - val_loss: 2.0574 - val_accuracy: 0.3137
Epoch 39/40
15/15 [==============================] - 0s 9ms/step - loss: 1.9938 - accuracy: 0.2769 - val_loss: 2.1215 - val_accuracy: 0.2157
Epoch 40/40
15/15 [==============================] - 0s 9ms/step - loss: 2.0602 - accuracy: 0.2484 - val_loss: 2.2229 - val_accuracy: 0.1961
```

Fig 4.6: Loss and Accuracy at the end of training

Similarly we have a plot for training loss Vs validation loss of ANN,

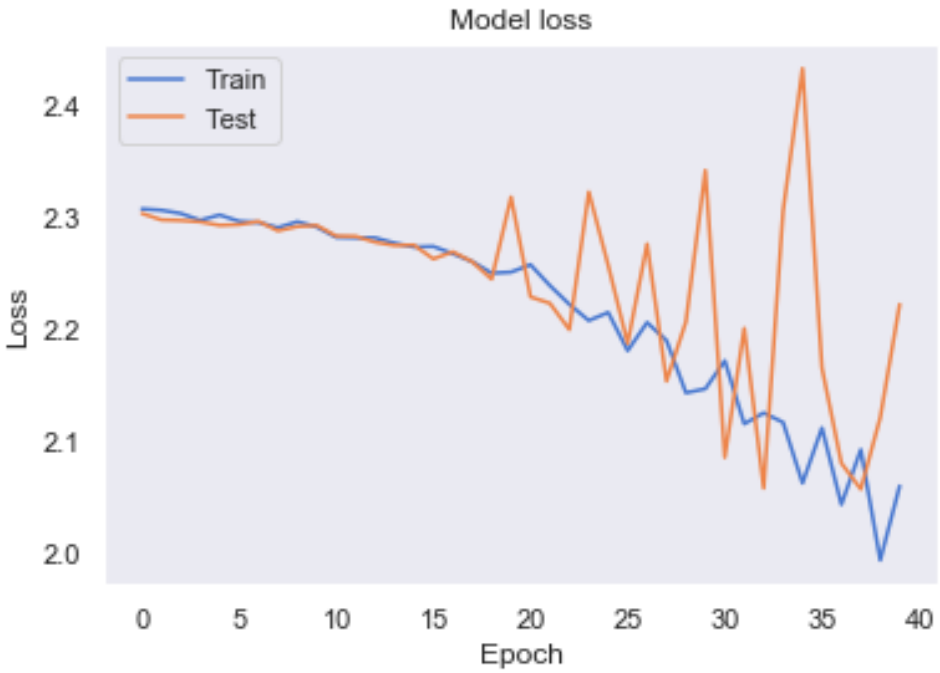

Fig 4.7: Train Loss Vs Valid Loss for ANN

Also a plot for training accuracy Vs validation accuracy

Fig 4.8: Train Accuracy Vs Valid Accuracy

Similarly, the confusion matrix plot for ANN is below,

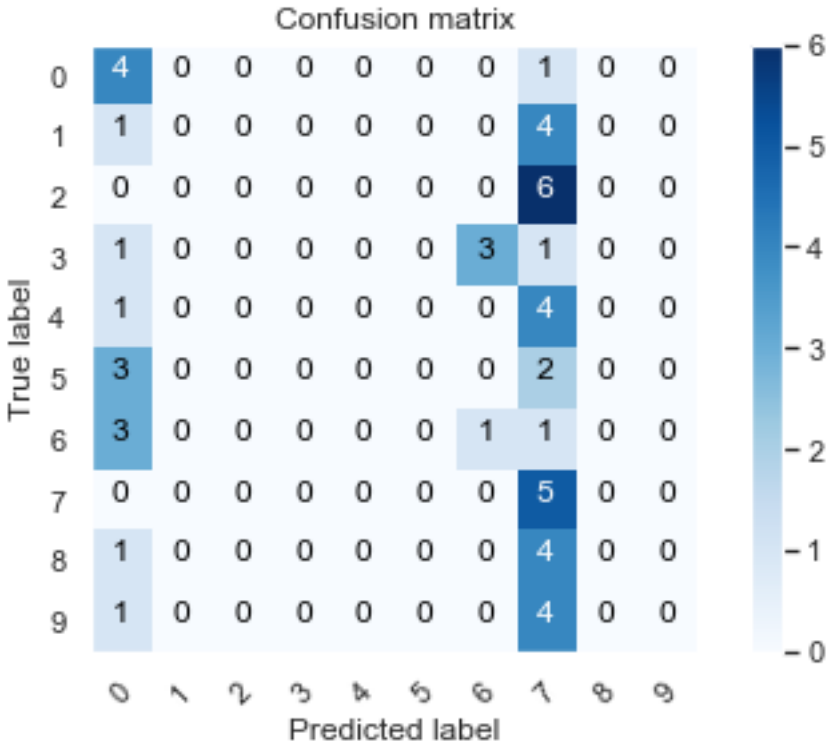

Fig 4.9: Confusion matrix plot for ANN model

4.4 Significance of the results obtained

After training both the deep learning models with the designated parameters, we compared their final loss values, and also predicted some of the outputs using both of these models and analyzed both the obtained results.

For CNN model:

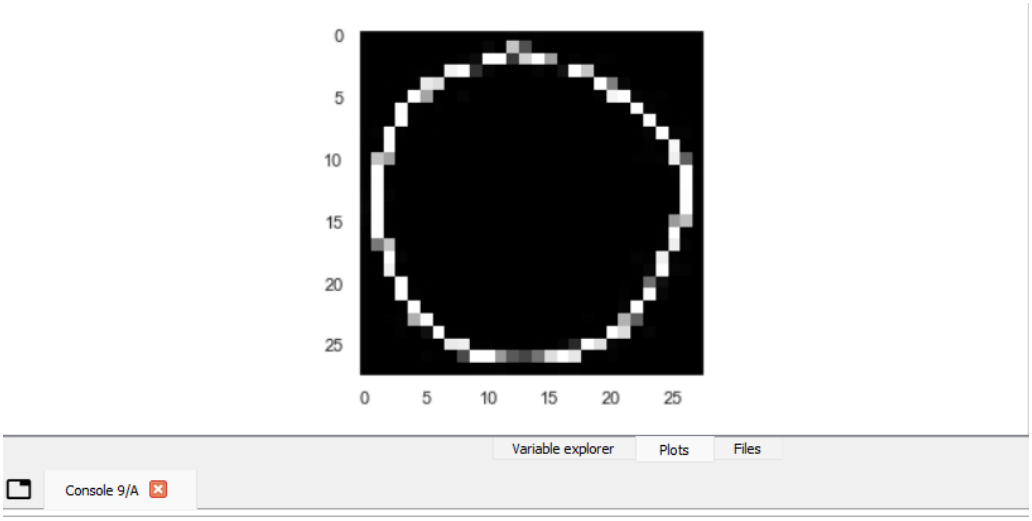

Fig 5.0: Output from CNN model

For ANN model:

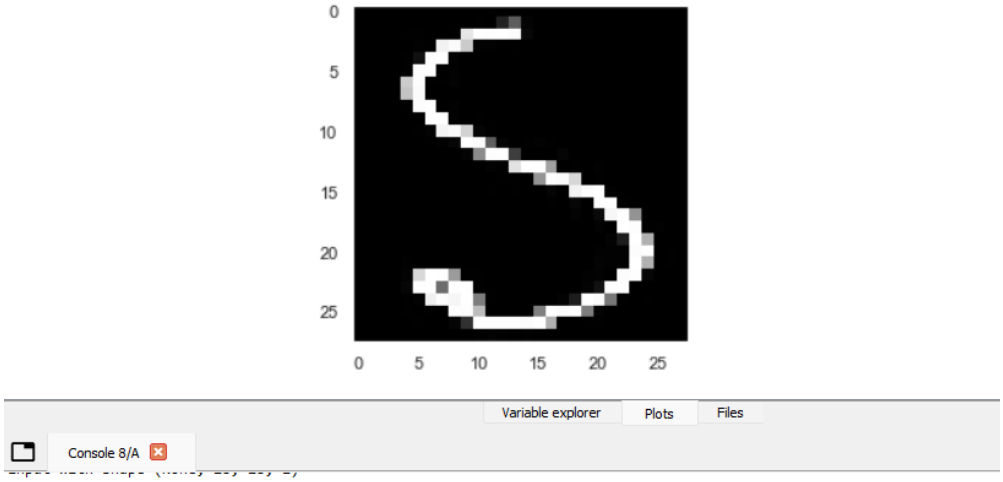

Fig 5.1 : Output from ANN model

4.4.1 Multi Digit Detection Using CNN Model

Input to the CNN model

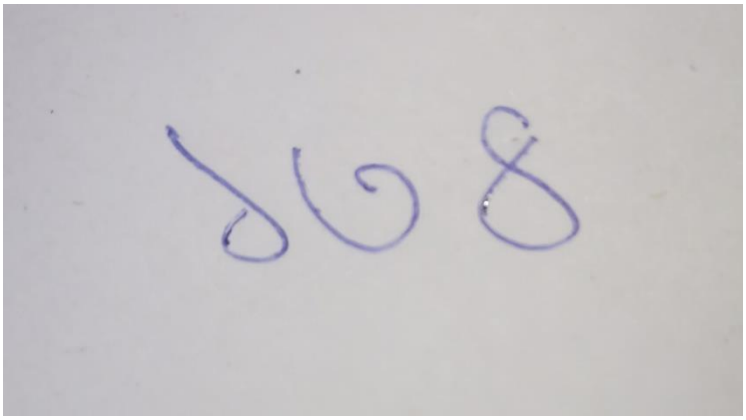

Fig 5.2: Handwritten assamese digit

After Image processing

Fig 5.3: Image after binary thresh holding

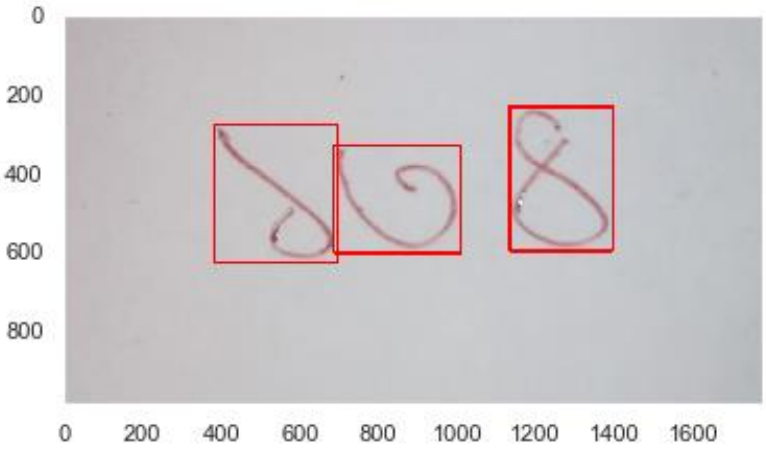

Fig 5.4: Image after applying contours

Now feeding the image to the CNN model

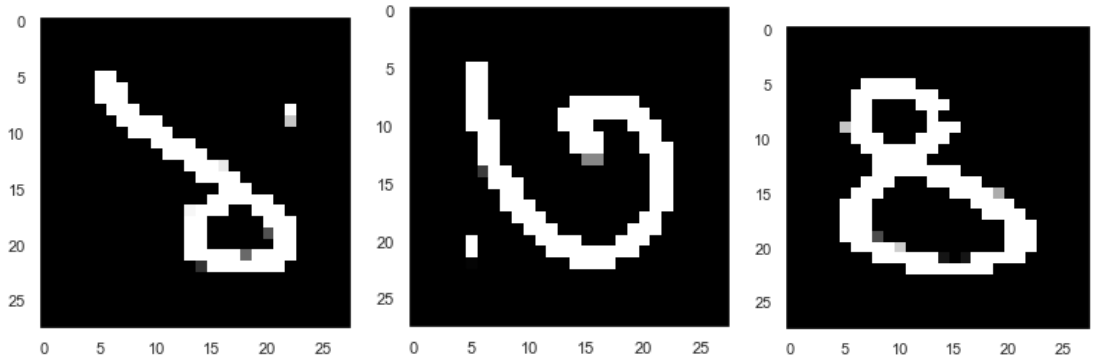

Fig 5.5: Cropped and reshaped images from original image

```
Prediction of digit number 1 : 3

---------------------------------------


Prediction of digit number 2 : 4

---------------------------------------


Prediction of digit number 3 : 4

---------------------------------------

========= FINAL PREDICTED NUMBER ============

The final predicted number is : 344
=============================================


In [19]:
```

Fig 5.6 Predicted Output from assamese digit image

We can see that the results predicted by the CNN model for the multi digit image were not accurate. This is because to detect digits from any handwritten image taken in any lighting condition would require a huge number of images for training. And also would require heavy computational power to train such a huge deep learning model with a large CNN architecture to do so. In the image taken by us, the pixel values in the image also matters very much. Therefore to build a robust model would require much more image processing and a huge assamese digit dataset.

CHAPTER 5
CONCLUSION AND FUTURE SCOPE OF WORK

In this chapter we will discuss the conclusion and the future scope of the project that we have carried out.

5.1 Summary of the work

Our project is to build a classifier from scratch that could classify handwritten assamese digits. We started this project by studying about digit detection and the required algorithms. During the study about algorithms, we studied about neural networks like Convolutional Neural Network (CNN) and Artificial Neural Network (ANN). Then we selected our Assamese Handwritten Digit dataset from IEEE dataport, by Prarthana Dutta. Then we tried different algorithms for our project, which were CNN and ANN. Then from these two models we selected the best. As our dataset is not huge and we were limited by both computing power and knowledge on deep learning, so building a sophisticated model was a challenging task.

5.2 Conclusions

So after training and analyzing the results from both the models, we can say that the CNN model is the best model in our case. The CNN model have lower validation accuracy with respect to validation accuracy in case of ANN and also it have higher validation accuracy of 92.16 % whereas the ANN model have a validation accuracy of 22.22 %.

5.3 Future scope of work

CNNs are very powerful. But task like detecting digits from rough pictures is very complicated and it takes a huge amount of data (would be couple of gigabytes). So a much more sophisticated model with huge amount of data along with more computation power might give better results. One may also go for transfer learning to get better results.

REFERENCES

[1]. Prarthana Dutta , Naresh Babu Muppalaneni, " DigiNet : Prediction of assamese handwritten digit using Convolutional Neural Network", in Wiley '05, 2021.

[2]. Yellapragada SS Bharadwaj , Rajaram P , Sriram V.P , Sudhakar S, Kolla Bhanu Prakash, " Effective Handwritten Digit Recognition using Deep Convolution Neural Network", in International Journal of Advanced Trends in Computer Science and Engineering, Volume No 9 No 2, 03-04, 2020.

ANNEXURES

Code Used:

For CNN implemention:

```
#import libraries
import pandas as pd
import numpy as np
from  tqdm.notebook import tqdm
from tensorflow.keras.preprocessing.image import img_to_array,load_img
import tensorflow as tf
import matplotlib.pyplot as plt
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import torch
import os
import cv2
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.utils import to_categorical
from tensorflow.keras import layers
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import itertools
from sklearn.metrics import *
from sklearn.model_selection import train_test_split
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.image as img
sns.set(style="dark",context="notebook",palette="muted")

#Tensorflow Version
print("TensorFlow Version:   "+tf.version.VERSION)
print("Keras Version:   "+tf.keras.__version__)

img_path = 'Data/train/'
df = pd.read_csv("Data/Train.csv")

X = np.array([img_to_array(load_img(img_path + df['filename'][i],target_size = (28,28,1),
```

```
grayscale = True))
        for i in tqdm(range(df.shape[0]))

        ]).astype('float32')


y = df['label']

print(X.shape,y.shape)

print(X)

#Exploratory Data Analysis

img_index = 0
print(y_train[img_index])
plt.imshow(X_train[img_index].reshape(28,28),cmap='Greys')

img_index = 3
print(y[img_index])
plt.imshow(X[img_index].reshape(28,28),cmap='Greys')



#Train test split
X_train,X_test,y_train,y_test = train_test_split(X,y, test_size=0.1,random_state = 2,stratify =
np.array(y))

print(X_train.shape)
print(y_train.shape)

#Normalization

print(X_train[0])


X_train /= 255
X_test /= 255
```

```
#onehot encoding # Label Encoding
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

#Model

model = tf.keras.Sequential()

model.add(layers.Conv2D(filters=32, kernel_size=(5,5), padding='Same',
                 activation=tf.nn.relu, input_shape = (28,28,1)))
model.add(layers.Conv2D(filters=32, kernel_size=(5,5), padding='Same',
                 activation=tf.nn.relu))
model.add(layers.MaxPool2D(pool_size=(2,2)))
model.add(layers.Dropout(0.25))


model.add(layers.Conv2D(filters=64, kernel_size=(3,3), padding='Same',
                 activation=tf.nn.relu, input_shape = (28,28,1)))
model.add(layers.Conv2D(filters=64, kernel_size=(3,3), padding='Same',
                 activation=tf.nn.relu))
model.add(layers.MaxPool2D(pool_size=(2,2),strides=(2,2)))
model.add(layers.Dropout(0.25))

model.add(layers.Flatten())
model.add(layers.Dense(256,activation=tf.nn.relu))
model.add(layers.Dropout(0.25))
model.add(layers.Dense(10,activation=tf.nn.softmax))



optimizer = tf.keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=1e-08, decay=0.0)


model.compile(optimizer = optimizer, loss='categorical_crossentropy',
          metrics=["accuracy"])
```

```
model.summary()

learning_rate_reduction = tf.keras.callbacks.ReduceLROnPlateau(monitor='val_acc',
                                    patience=3,
                                    verbose=1,
                                    factor=0.5,
                                    min_lr=0.00001)

epochs=40
batch_size = 32



datagen = ImageDataGenerator(
        featurewise_center=False,  # set input mean to 0 over the dataset
        samplewise_center=False,  # set each sample mean to 0
        featurewise_std_normalization=False,  # divide inputs by std of the dataset
        samplewise_std_normalization=False,  # divide each input by its std
        zca_whitening=False,  # apply ZCA whitening
        rotation_range=10,  # randomly rotate images in the range (degrees, 0 to 180)
        zoom_range = 0.1, # Randomly zoom image
        width_shift_range=0.1,  # randomly shift images horizontally (fraction of total width)
        height_shift_range=0.1,  # randomly shift images vertically (fraction of total height)
        horizontal_flip=False,  # randomly flip images
        vertical_flip=False)  # randomly flip images


datagen.fit(X_train)




# checking CUDA availability
if(tf.test.is_built_with_cuda() == True):
```

```
    print("CUDA Available.. Just wait a few moments...")
else:
    print("CUDA not Available.. May the force be with you.")

#training

history = model.fit_generator(datagen.flow(X_train,y_train, batch_size=batch_size),
                              epochs = epochs, validation_data = (X_test,y_test),
                              verbose = 2, steps_per_epoch=X_train.shape[0] // batch_size
                              , callbacks=[learning_rate_reduction])

#saving model
model.save('newmodel.h5')

#testing
image_index = 3
# print("Original output:",y_test[image_index])
plt.imshow(X_test[image_index].reshape(28,28), cmap='Greys')
pred = model.predict(X_test[image_index].reshape(1,28,28,1))
print("Predicted output:", pred.argmax())



fig, ax = plt.subplots(2,1)
ax[0].plot(history.history['loss'], color='b', label="Training loss")
ax[0].plot(history.history['val_loss'], color='r', label="validation loss",axes =ax[0])
legend = ax[0].legend(loc='best', shadow=True)

ax[1].plot(history.history['accuracy'], color='b', label="Training accuracy")
ax[1].plot(history.history['val_accuracy'], color='r',label="Validation accuracy")
legend = ax[1].legend(loc='best', shadow=True)



def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
```

```
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j],
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

# Predict the values from the validation dataset
Y_pred = model.predict(X_test)
# Convert predictions classes to one hot vectors
Y_pred_classes = np.argmax(Y_pred,axis = 1)
# Convert validation observations to one hot vectors
Y_true = np.argmax(y_test,axis = 1)
# compute the confusion matrix
confusion_mtx = confusion_matrix(Y_true, Y_pred_classes)
# plot the confusion matrix
plot_confusion_matrix(confusion_mtx, classes = range(10))




# Errors are difference between predicted labels and true labels
errors = (Y_pred_classes - Y_true != 0)

Y_pred_classes_errors = Y_pred_classes[errors]
```

```
Y_pred_errors = Y_pred[errors]
Y_true_errors = Y_true[errors]
X_val_errors = X_test[errors]

def display_errors(errors_index,img_errors,pred_errors, obs_errors):
    """ This function shows 6 images with their predicted and real labels"""
    n = 0
    nrows = 2
    ncols = 3
    fig, ax = plt.subplots(nrows,ncols,sharex=True,sharey=True)
    for row in range(nrows):
        for col in range(ncols):
            error = errors_index[n]
            ax[row,col].imshow((img_errors[error]).reshape((28,28)))
            ax[row,col].set_title(" Predicted :{} True
:{}".format(pred_errors[error],obs_errors[error]))
            n += 1

# Probabilities of the wrong predicted numbers
Y_pred_errors_prob = np.max(Y_pred_errors,axis = 1)

# Predicted probabilities of the true values in the error set
true_prob_errors = np.diagonal(np.take(Y_pred_errors, Y_true_errors, axis=1))

# Difference between the probability of the predicted label and the true label
delta_pred_true_errors = Y_pred_errors_prob - true_prob_errors

# Sorted list of the delta prob errors
sorted_dela_errors = np.argsort(delta_pred_true_errors)

# Top 6 errors
most_important_errors = sorted_dela_errors[-6:]

# Show the top 6 errors
display_errors(most_important_errors, X_val_errors, Y_pred_classes_errors, Y_true_errors)



#testing
```

```
kernel = np.ones((10,10),np.uint8)
kernel1 = np.ones((1,1),np.uint8)



image = cv2.imread('./test11.jpg')
grey = cv2.cvtColor(image.copy(), cv2.COLOR_BGR2GRAY)
ret, thresh = cv2.threshold(grey.copy(), 170, 255, cv2.THRESH_BINARY_INV)

thresh = cv2.erode(thresh,kernel1,iterations=1)
thresh = cv2.dilate(thresh,kernel,iterations=2)
plt.imshow(thresh,cmap='gray')
plt.show()
contours, _ = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
preprocessed_digits = []
i = 0
for c in contours:
    x,y,w,h = cv2.boundingRect(c)
    if ( w <= thresh.shape[0]*(0.05)) and h <= (thresh.shape[1]*(0.05)): # Even after we have
eroded and dilated the images, we will still take
                                                    #precaution not to have small unwanted pixels or
features being detected


        continue;
    else:


        # Creating a rectangle around the digit in the original image (for displaying the digits fetched
via contours)
        cv2.rectangle(image, (x,y), (x+w, y+h), color=(255, 0, 0), thickness=5)

        # Cropping out the digit from the image corresponding to the current contours in the for loop
        digit = thresh[y:y+h, x:x+w]



        # Resizing that digit to (18, 18)
        i +=1
        resized_digit = cv2.resize(digit, (18,18))

        # Padding the digit with 5 pixels of black color (zeros) in each side to finally produce the
```

```
image of (28, 28)
        padded_digit = np.pad(resized_digit, ((5,5),(5,5)), "constant", constant_values=0)

        # Adding the preprocessed digit to the list of preprocessed digits
        preprocessed_digits.append(padded_digit)
print("\n\n\n----------Contoured Image----------")
print('number of digits in the number : ',i)
plt.imshow(image, cmap="gray")
plt.show()

inp = np.array(preprocessed_digits)


from tensorflow.keras.models import load_model

model = load_model('newmodel.h5')


final_num = 'The final predicted number is : '
index = 0
for digit in preprocessed_digits:
    prediction = model.predict(digit.reshape(1, 28, 28, 1))   # the first 1 signifies the batch size, the last 1 signifies greyscale image
    plt.imshow(digit.reshape(28, 28), cmap="gray")
    plt.show()
    print("\n\n Prediction of digit number {} : {}".format(index+1,np.argmax(prediction)))
    index +=1
    final_num += str(np.argmax(prediction))
    print ("\n\n---------------------------------------\n\n")
print ("========= FINAL PREDICTED NUMBER ========== \n")
print(final_num)
print ("========================================== \n\n")

model
```

For ANN implementation:

```
#import libraries
import pandas as pd
import numpy as np
from  tqdm.notebook import tqdm
from tensorflow.keras.preprocessing.image import img_to_array,load_img
import tensorflow as tf
import matplotlib.pyplot as plt
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import torch
import os
import cv2
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.utils import to_categorical
from tensorflow.keras import layers
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import itertools
from sklearn.metrics import *
from sklearn.model_selection import train_test_split
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.image as img
sns.set(style="dark",context="notebook",palette="muted")

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout

img_path = 'Data/train/'
df = pd.read_csv("Data/Train.csv")

X = np.array([img_to_array(load_img(img_path + df['filename'][i],target_size = (28,28,1),
grayscale = True))
        for i in tqdm(range(df.shape[0]))
```

```
]).astype('float32')


y = df['label']



#Exploratory Data Analysis

img_index = 0
print(y_train[img_index])
plt.imshow(X_train[img_index].reshape(28,28),cmap='Greys')

img_index = 3
print(y[img_index])
plt.imshow(X[img_index].reshape(28,28),cmap='Greys')



#Train test split
X_train,X_test,y_train,y_test = train_test_split(X,y, test_size=0.1,random_state = 42,stratify =
np.array(y))

X_train /= 255
X_test /= 255

#onehot encoding # Label Encoding
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

X_train = X_train.reshape(-1,784)
X_test = X_test.reshape(-1,784)

#ANN
classifier = Sequential()

# Adding the input layer and the first hidden layer
classifier.add(Dense(units = (784), activation = 'relu', input_dim = 784))
```

```
# Adding the second hidden layer
classifier.add(Dense(units = 128, activation = 'relu'))
classifier.add(Dense(units = 128, activation = 'relu'))
classifier.add(Dropout(0.2))

classifier.add(Dense(units = 64, activation = 'relu'))
classifier.add(Dense(units = 64, activation = 'relu'))
classifier.add(Dropout(0.2))

classifier.add(Dense(units = 32, activation = 'relu'))
classifier.add(Dense(units = 32, activation = 'relu'))
classifier.add(Dropout(0.05))


# Adding the output layer
classifier.add(Dense(units = 10, activation = 'softmax'))


# Compiling the ANN
classifier.compile(optimizer = 'sgd', loss = 'categorical_crossentropy', metrics = ['accuracy'])

classifier.summary()

# Fitting the ANN to the Training set
history = classifier.fit(X_train, y_train,
                         validation_data = (X_test, y_test),
                         batch_size = 32,
                         epochs = 40)

# Plot training & validation accuracy values
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```

```
# Plot training & validation loss values
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()


def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j],
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

# Predict the values from the validation dataset
Y_pred = classifier.predict(X_test)
```

```
# Convert predictions classes to one hot vectors
Y_pred_classes = np.argmax(Y_pred,axis = 1)
# Convert validation observations to one hot vectors
Y_true = np.argmax(y_test,axis = 1)
# compute the confusion matrix
confusion_mtx = confusion_matrix(Y_true, Y_pred_classes)
# plot the confusion matrix
plot_confusion_matrix(confusion_mtx, classes = range(10))

classifier.save('annmodel.hf')



#testing
image_index = 3
# print("Original output:",y_test[image_index])
plt.imshow(X_test[image_index].reshape(28,28), cmap='Greys')
pred = classifier.predict(X_test[image_index].reshape(1,784))
print("Predicted output:", pred.argmax())
```