ESE-588 Pattern Recognition
Spring 2018

Project: Predicting Handwritten Alphabets

Due Date: 5/16/18

Sahil Jindal 111471177

Ameya Lokre 111675524

Kislay Kislay 111733767

1 Introduction

The goal of our project is to build a web application to predict the input characters written into an HTML canvas real time. We aim to correctly characterize the digits from 0-9 and the capital[A-Z] and small[a-z] English alphabets. If we are able to correctly classify these alphabets and digits, it can be a first step in digitalization of the handwritten records.

We use the EMNIST dataset which is an extended version of the standard MNIST dataset that contains alphabets as well as the digits. The EMNIST dataset is a set of handwritten character digits derived from the NIST Special Database 19 and converted to a 28x28 pixel image format and dataset structure that directly matches the MNIST dataset. The dataset is provided in two file formats. Both versions of the dataset contain identical information, and are provided entirely for the sake of convenience. The first dataset is provided in a Matlab format that is accessible through both Matlab and Python (using the scipy.io.loadmat function). The second version of the dataset is provided in the same binary format as the original MNIST dataset. We used the mat format for the purpose of our project as it can be easily read and manipulated using python.

There are six different splits provided in this dataset. A short summary of the dataset is provided below:

1. EMNIST ByClass: 814,255 characters. 62 unbalanced classes.
2. EMNIST ByMerge: 814,255 characters. 47 unbalanced classes.
3. EMNIST Balanced: 131,600 characters. 47 balanced classes.
4. EMNIST Letters: 145,600 characters. 26 balanced classes.
5. EMNIST Digits: 280,000 characters. 10 balanced classes.
6. EMNIST MNIST: 70,000 characters. 10 balanced classes.

For the purpose of this project we used the EMNIST Balanced dataset where 112800 images were used to train the data and 18800 were used for testing the model.

The project will be realized using tensorflowjs which is the javascript version of the tensorflow python API by google. tensorflowjs has been announced recently during the tensorflow event 2018. It provides a way of integrating the machine learning capabilities into our browser. We can train the models in the browser and also import already trained models and then use them for prediction.

We will build a convolutional neural network(CNN) to build a classifier using python and then convert it into a JSON model that can be used by tensorflowjs for making predictions.

2 Techniques and Tools

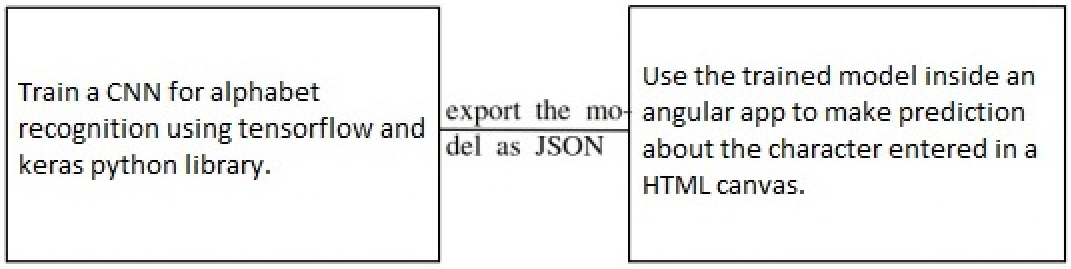

Train a CNN for alphabet recognition using tensorflow and keras python library.
export the model as JSON
Use the trained model inside an angular app to make prediction about the character entered in a HTML canvas.

Figure 1: High level description

The approach for this project can be divided into 2 main parts. First is creating a CNN using the tensorflow and keras library to train a model which classifies the images as one of the 47 classes and second is building an angular app that has an HTML canvas element which can be used to take input and then using that to make predictions.

2.1 CNN Classifier

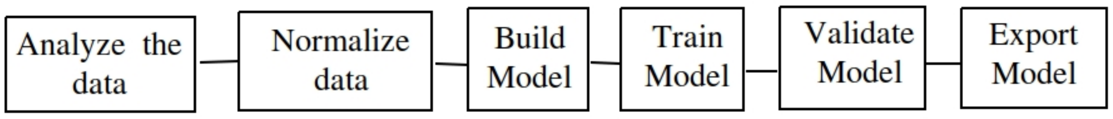


Figure 2: Training the model

We train a CNN classifier to predict the input alphabet. Here are the basic steps that we followed to build this CNN

1. **Analyze Data** The first step in building the model to analyze and understand the data. Our input data is a mat file that has 112800 training images and 18800 testing images. The output is one of the 47 classes and mapping of these classes to the corresonding ASCII values is also given in the dataset. The images are 28 × 28 black and white images. These are flattened out and need to put into 28 × 28 × 1 image.

2. **Normalize Data** The pixel values in image range from 0-255 and historically it has been noted that this data doesn't work good with CNN. So we normalize the dataset by dividing the pixel vales by 255. Also the input data images are rotated to we need to put them in correct orientation before we train our classifier.

3. **Build Model** The next step is to build a model. Our model has 8 layers. The first 2 are 2-D convolution layers followed by a max pooling layer. Then in order to prevent over-fitting of data we dropout some of our nodes. Here we from 25% of out nodes. Then we add a relu activation layer with 50% dropout. Finally we add a softmax layer to get the output probabilities for each class.

4. **Train Model** We train the above model using our training and testing data and obtain the weights and biases for our neural network. We use 10 epochs to train our classifier with a batch size of 256.

5. **Export Model** Once we have a trained model tensorflow provides an API to convert your h5 model into a JSON model that can be consumed by the tensorflowjs and can be used to make predictions.

2.2 Angular App

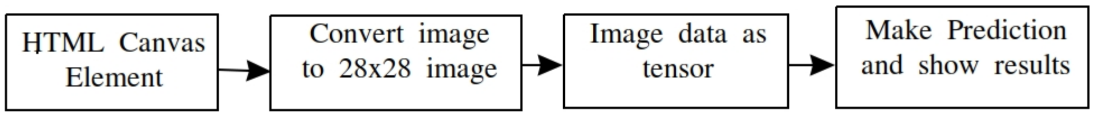

HTML Canvas Element
Convert image to 28x28 image
Image data as tensor
Make Prediction and show results

Figure 3: Angular App

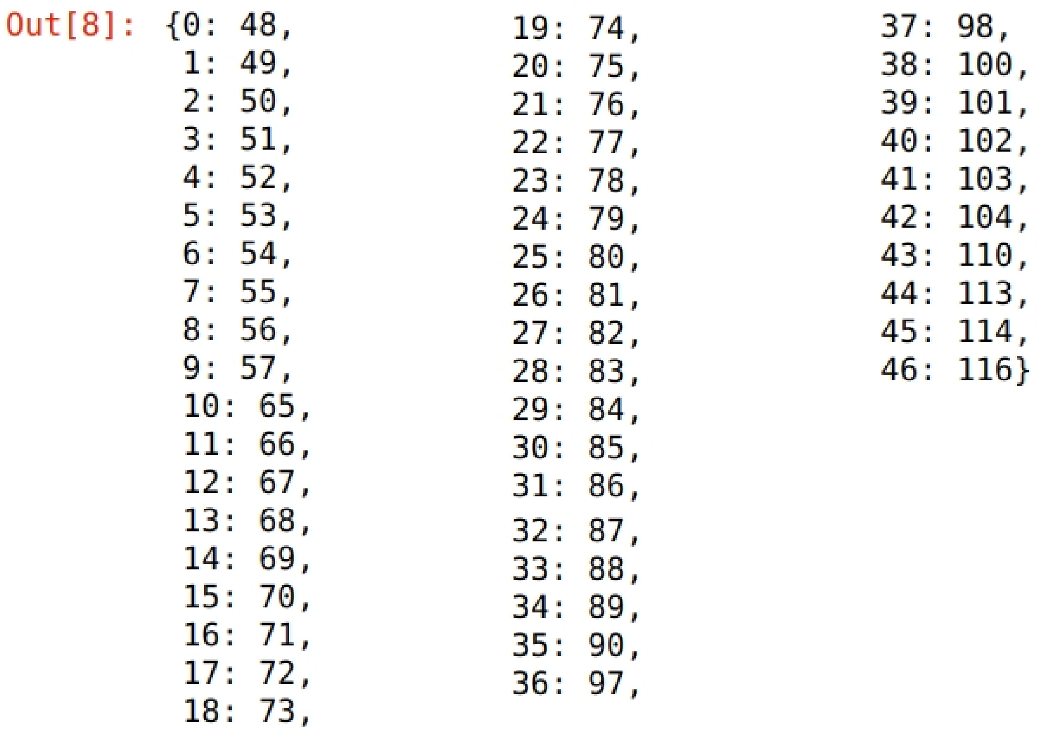
For the web app we build a barebone angular app using angular CLI that has an HTML canvas component which can be used to take the input and predictions are made based on this input. Following are the steps involved in this process

1. **Canvas Component** To take the input we use the standard HTML5 canvas component which can be used to take touch, pen and mouse input to draw on an HTML page.
2. **Convert Image to 28 $\times$ 28** 28 $\times$ 28 is a very tiny drawing board and it is not practical to take such small input. So we have a 300 $\times$ 300 size canvas which is compressed to a 28 $\times$ 28 image.
3. **Image as a Tensor** tensorflowjs provides a method to convert the image data into a tensor. The image data is a 28 $\times$ 28 $\times$ 1 input but we need a 1 $\times$ 28 $\times$ 28 $\times$ 1 input for classifcation so tensor needs to be reshaped into required dimension.
4. **Make Prediction** The final step is to load the pre-trained model and make predictions using the input captured using above procedure.

3 Results and Verification

As mentioned earlier, we used the balanced-emnist dataset in mat format. The dataset has 112800 training images and 18800 test images. The images are 28 $\times$ 28 $\times$ 1 images that means there is no rgb component and the images are black and white.

There are 47 classes that the images have been classified into the corresponding ASCII value mappings have been listed in the image below:



```
Out[8]: {0: 48,          19: 74,          37: 98,
         1: 49,          20: 75,          38: 100,
         2: 50,          21: 76,          39: 101,
         3: 51,          22: 77,          40: 102,
         4: 52,          23: 78,          41: 103,
         5: 53,          24: 79,          42: 104,
         6: 54,          25: 80,          43: 110,
         7: 55,          26: 81,          44: 113,
         8: 56,          27: 82,          45: 114,
         9: 57,          28: 83,          46: 116}
         10: 65,         29: 84,
         11: 66,         30: 85,
         12: 67,         31: 86,
         13: 68,         32: 87,
         14: 69,         33: 88,
         15: 70,         34: 89,
         16: 71,         35: 90,
         17: 72,         36: 97,
         18: 73,
```

Figure 4: ASCII Mappings

The images are initially mirrored and rotated by 90 degrees, so we need to process the images before we can train our model. The sample image before pre-processing is

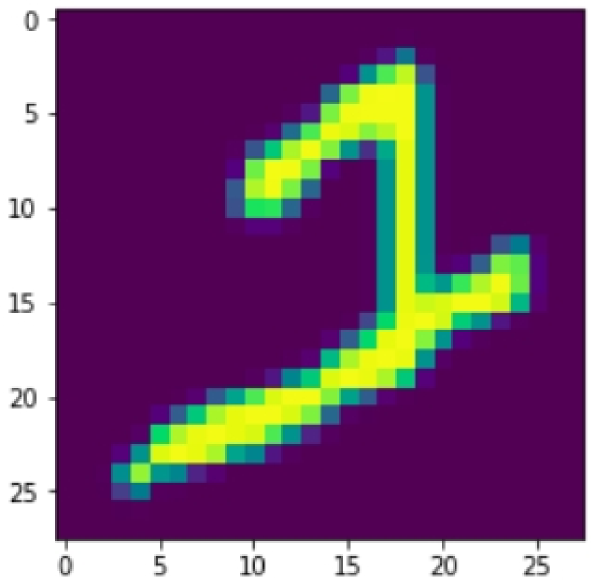


Figure 5: Before Pre-processing

The same image after processing is

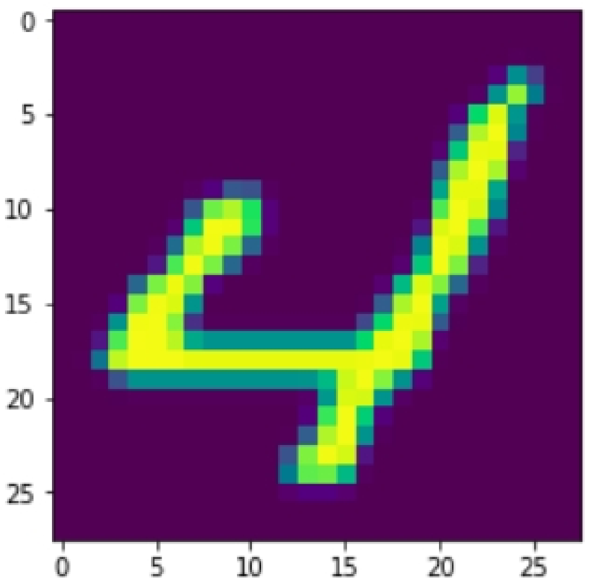
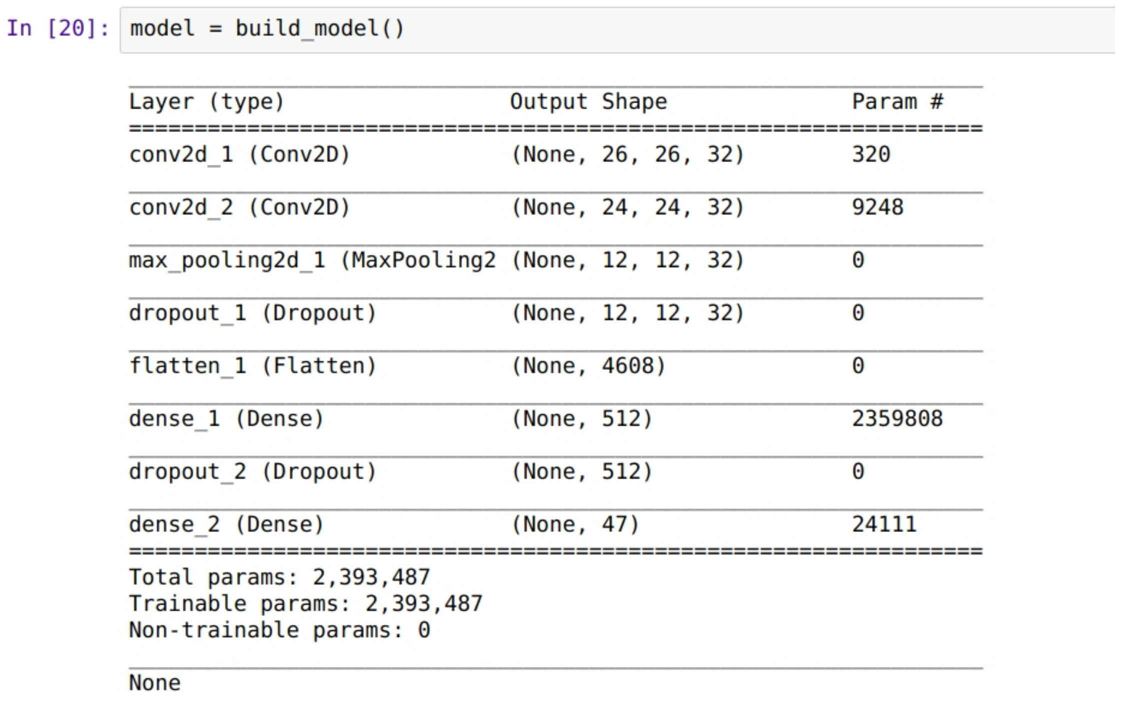


Figure 6: After Pre-processing

We create a model using the keras library and tensorflow as the backend. The summary of the model is

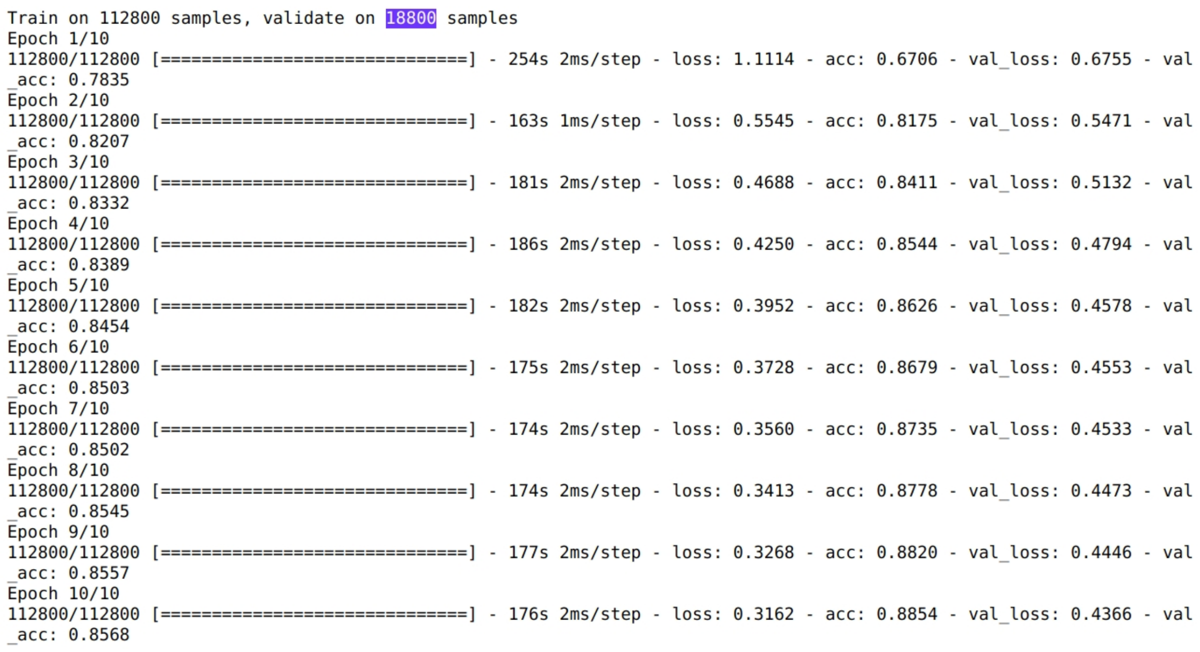


```
In [20]: model = build_model()

_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320       
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 24, 24, 32)        9248      
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 12, 12, 32)        0         
_________________________________________________________________
dropout_1 (Dropout)          (None, 12, 12, 32)        0         
_________________________________________________________________
flatten_1 (Flatten)          (None, 4608)              0         
_________________________________________________________________
dense_1 (Dense)              (None, 512)               2359808   
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0         
_________________________________________________________________
dense_2 (Dense)              (None, 47)                24111     
=================================================================
Total params: 2,393,487
Trainable params: 2,393,487
Non-trainable params: 0
_________________________________________________________________
None
```

Figure 7: Model

We then trained the model using our training dataset and validated the same using the test set. We used 10 epochs and the results were as follows:



```
Train on 112800 samples, validate on 18800 samples
Epoch 1/10
112800/112800 [==============================] - 254s 2ms/step - loss: 1.1114 - acc: 0.6706 - val_loss: 0.6755 - val
_acc: 0.7835
Epoch 2/10
112800/112800 [==============================] - 163s 1ms/step - loss: 0.5545 - acc: 0.8175 - val_loss: 0.5471 - val
_acc: 0.8207
Epoch 3/10
112800/112800 [==============================] - 181s 2ms/step - loss: 0.4688 - acc: 0.8411 - val_loss: 0.5132 - val
_acc: 0.8332
Epoch 4/10
112800/112800 [==============================] - 186s 2ms/step - loss: 0.4250 - acc: 0.8544 - val_loss: 0.4794 - val
_acc: 0.8389
Epoch 5/10
112800/112800 [==============================] - 182s 2ms/step - loss: 0.3952 - acc: 0.8626 - val_loss: 0.4578 - val
_acc: 0.8454
Epoch 6/10
112800/112800 [==============================] - 175s 2ms/step - loss: 0.3728 - acc: 0.8679 - val_loss: 0.4553 - val
_acc: 0.8503
Epoch 7/10
112800/112800 [==============================] - 174s 2ms/step - loss: 0.3560 - acc: 0.8735 - val_loss: 0.4533 - val
_acc: 0.8502
Epoch 8/10
112800/112800 [==============================] - 174s 2ms/step - loss: 0.3413 - acc: 0.8778 - val_loss: 0.4473 - val
_acc: 0.8545
Epoch 9/10
112800/112800 [==============================] - 177s 2ms/step - loss: 0.3268 - acc: 0.8820 - val_loss: 0.4446 - val
_acc: 0.8557
Epoch 10/10
112800/112800 [==============================] - 176s 2ms/step - loss: 0.3162 - acc: 0.8854 - val_loss: 0.4366 - val
_acc: 0.8568
```

Figure 8: Training Results

Once we get a trained model we save it as a h5 file and then use the tensorflowjs API to convert it into a JSON file.

Next we created a bare-bone angular app using angular cli that has an HTML5 canvas component. We programmed this component to take mouse inputs. We then use the javascript functions to get a 28 × 28 representation of the image. Using tensorflowjs we load the pre-trained model and feed it the 28 × 28 image to make predictions. The following images are screenshots from our final app.

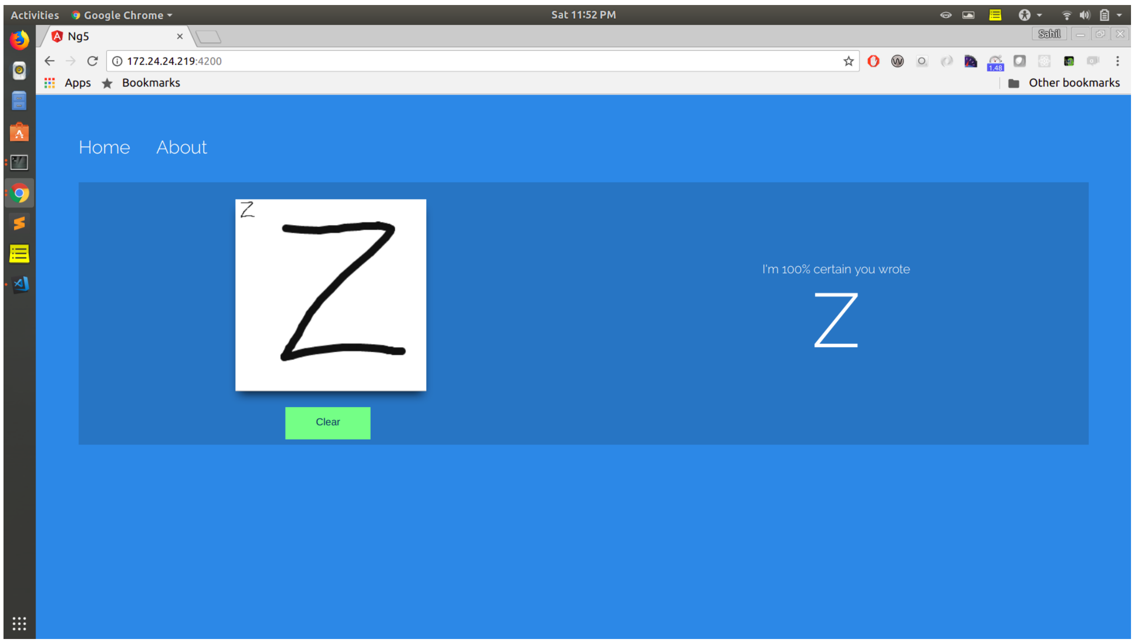


Figure 9: Z correctly predicted

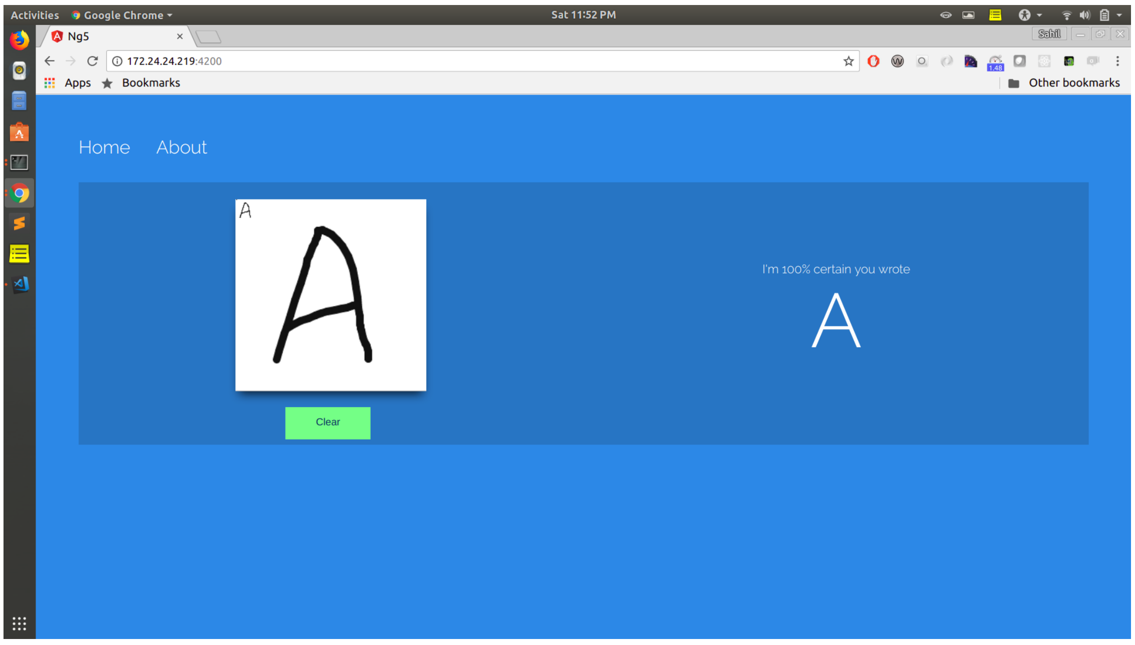


Figure 10: A correctly predicted

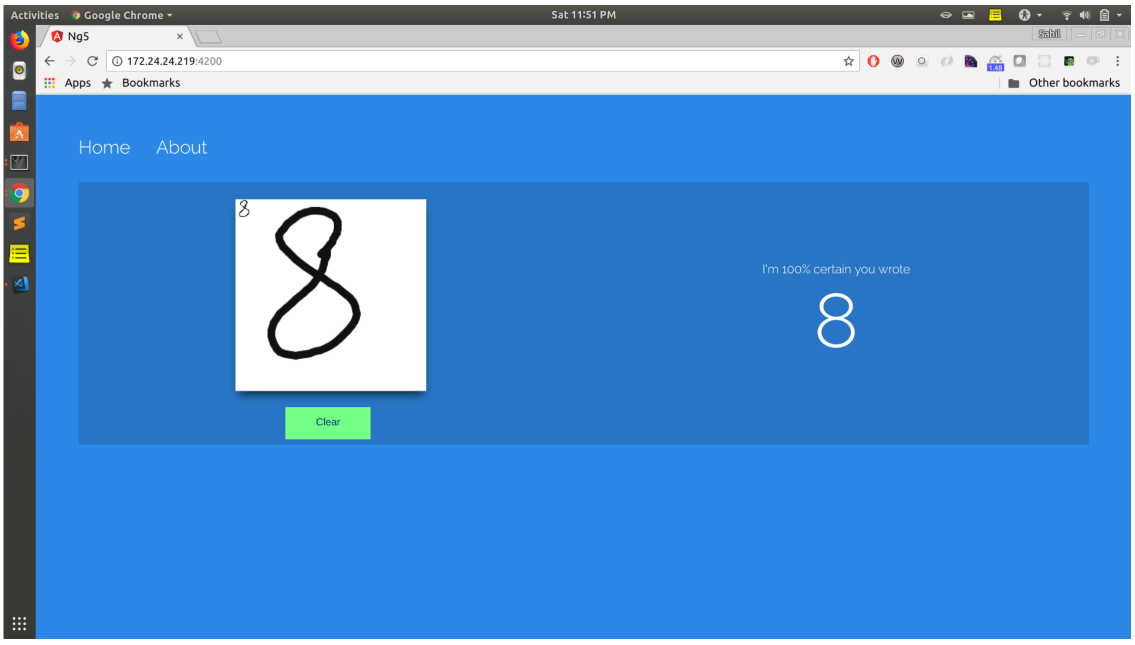
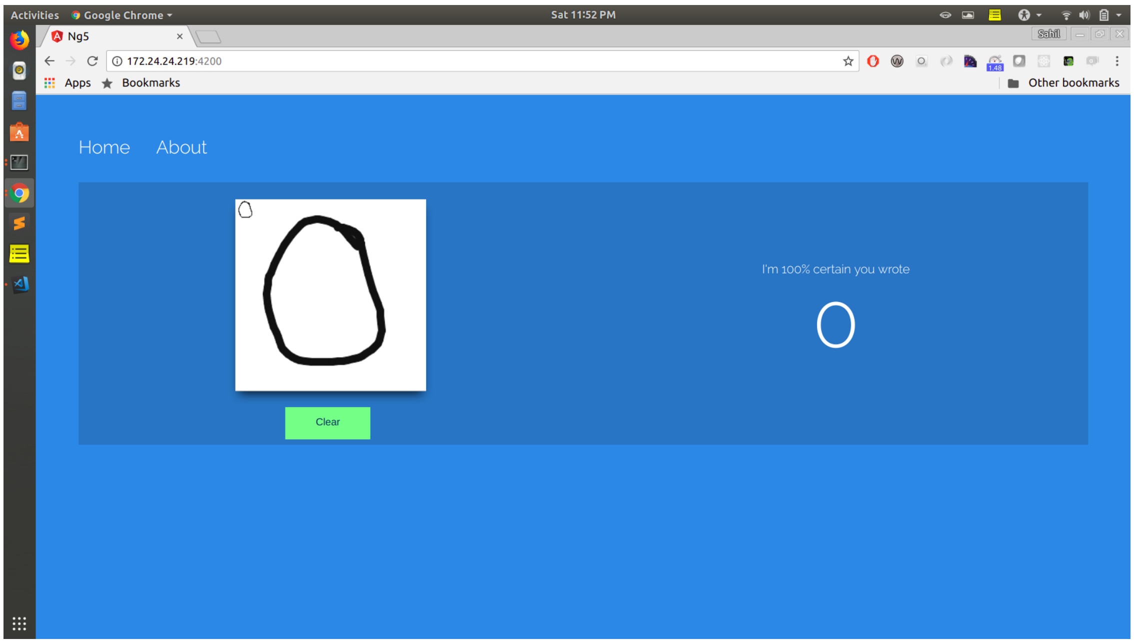


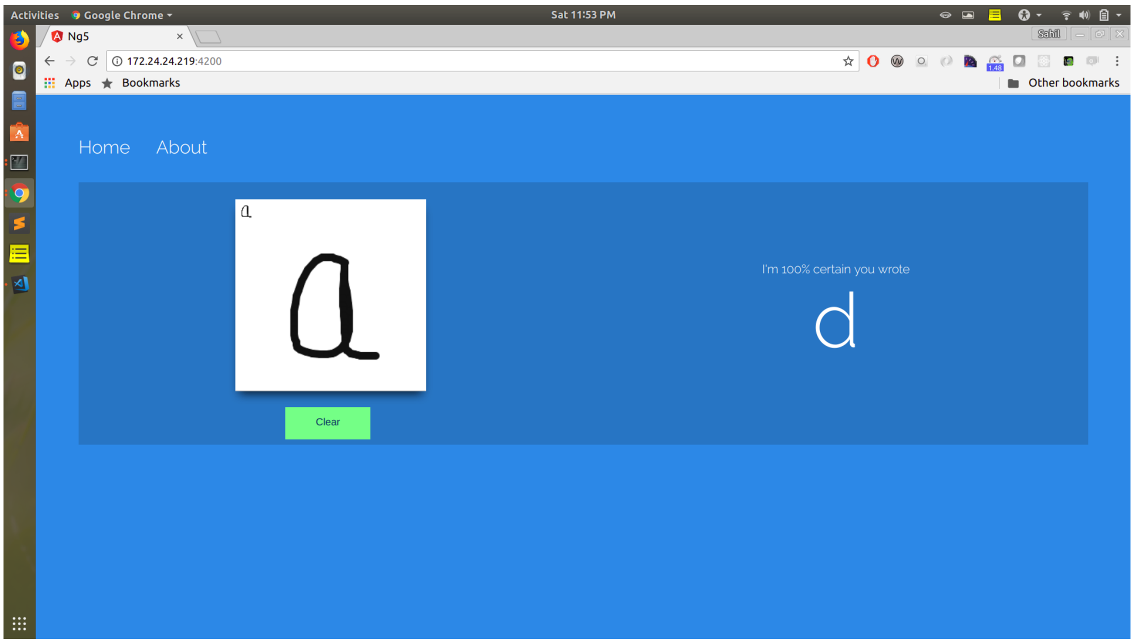
Figure 11: 8 correctly predicted



I'm 100% certain you wrote

O

Figure 12: O correctly predicted



I'm 100% certain you wrote

d

Figure 13: a wrongly predicted as d

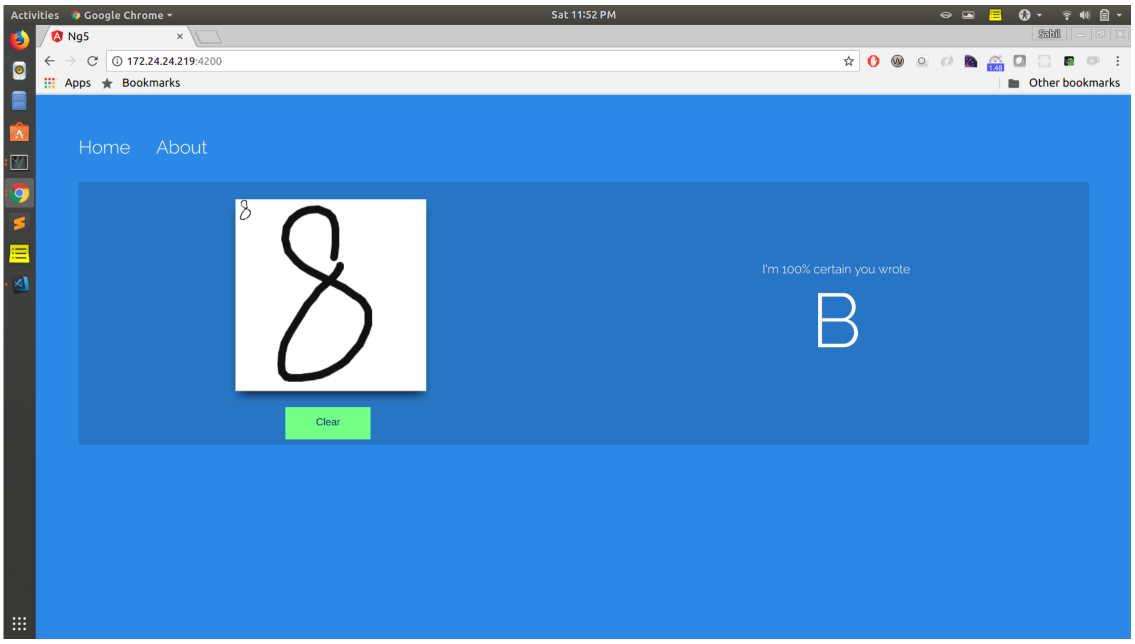


Figure 14: 8 wrongly predicted as B

4 Conclusion

Our webapp is able to capture the user input and predict what alphabet or digit did the user input. But the results are not 100% reliable. There are many instances when our classifiers predicts wrong alphabet. Future work would include trying out different classifiers and also we can make this model to learn from the user input. We can take user input to check if our prediction was correct and the model can learn if it predicted some alphabet incorrectly.

This idea can further be extended into a notes taking app that can use the pen input to read the data and convert your handwritten text into a digital document.

5 References

[1] https://js.tensorflow.org/
[2] https://www.kaggle.com/hardiksaraiya/cnn-emnist
[3] https://www.kaggle.com/dmborovoy/emnist-neural-network