IMPLEMENTATION OF HANDWRITTEN CHARACTER RECOGNITION BY TRAINING AND DEPLOYING A NEURAL NETWORK TO AN ANDROID APPLICATION

ARITRA KOLEY

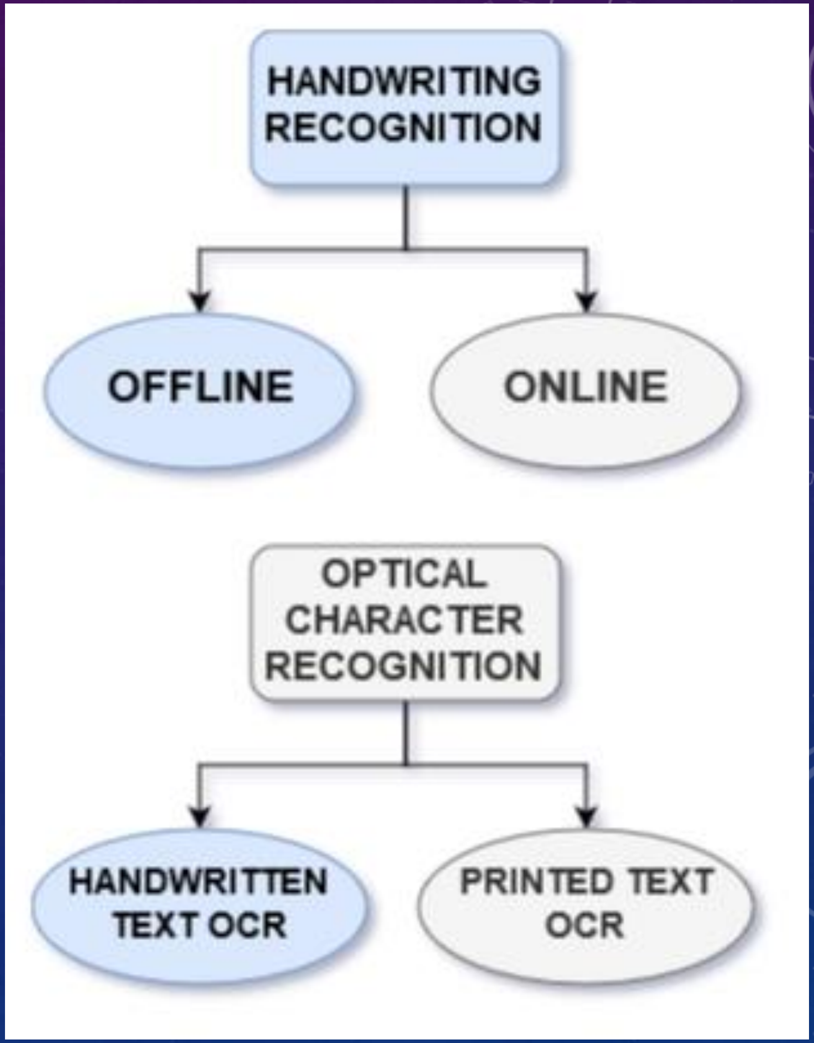
SUPERVISOR: DR. GAURAV BARANWAL

OUTLINE

- Introduction
  - Basic understanding of the functionality of the designed system
  - Schematics
- Demonstration
  - Full functionality demonstration
  - Example of a use case
- Implementation Details
  - Dataset details, Techniques used, Difficulties faced, etc.
- Conclusion and Future Work
- References

INTRODUCTION

- The simulation of the natural human reading process and interpretation of handwritten textual matter by machines has been a goal of computer scientists and natural language researchers for a long time.
- With the rise in computational capacity of computers, machine learning became the preferred technique for a lot of pattern recognition tasks, including handwriting recognition
- The next logical step for handwriting recognition functionality is to become available on mobile platforms.



CURRENT TRENDS:

- Machine learning has traditionally been implemented in large powerful computing environments.
- Only recently have the big names like Google, pushed towards local inference engines in mobile phones.
- Previously, the data collected on or through smartphones were off-loaded to the cloud to be analyzed.
- The trend now is shifting to be able to process the data collected using the device right on the device.
  - **Advantage**: Avoids many of the privacy and security risks involved with sending user data to a third-party cloud service provider.
- Keeping all that in mind, the aim of this project was to keep as much of the processing as is viable while handling user data limited to the device.

CURRENT TRENDS (contd.)

- Applications:
  - CamScanner: [Printed Text OCR] [Localized]
  - Evernote: [Handwriting Recognition both Offline and Online]
  - Microsoft OneNote: [Similar functionality with extended features for Online Recognition]
  - WritePad SDK: [Open source SDK for Online Handwriting Recognition]
  - Google Handwriting Keyboard: [Online Handwriting Recognition]
  - Text Scanner – Text Recognition – Text OCR: [Android App. Handwriting OCR.]

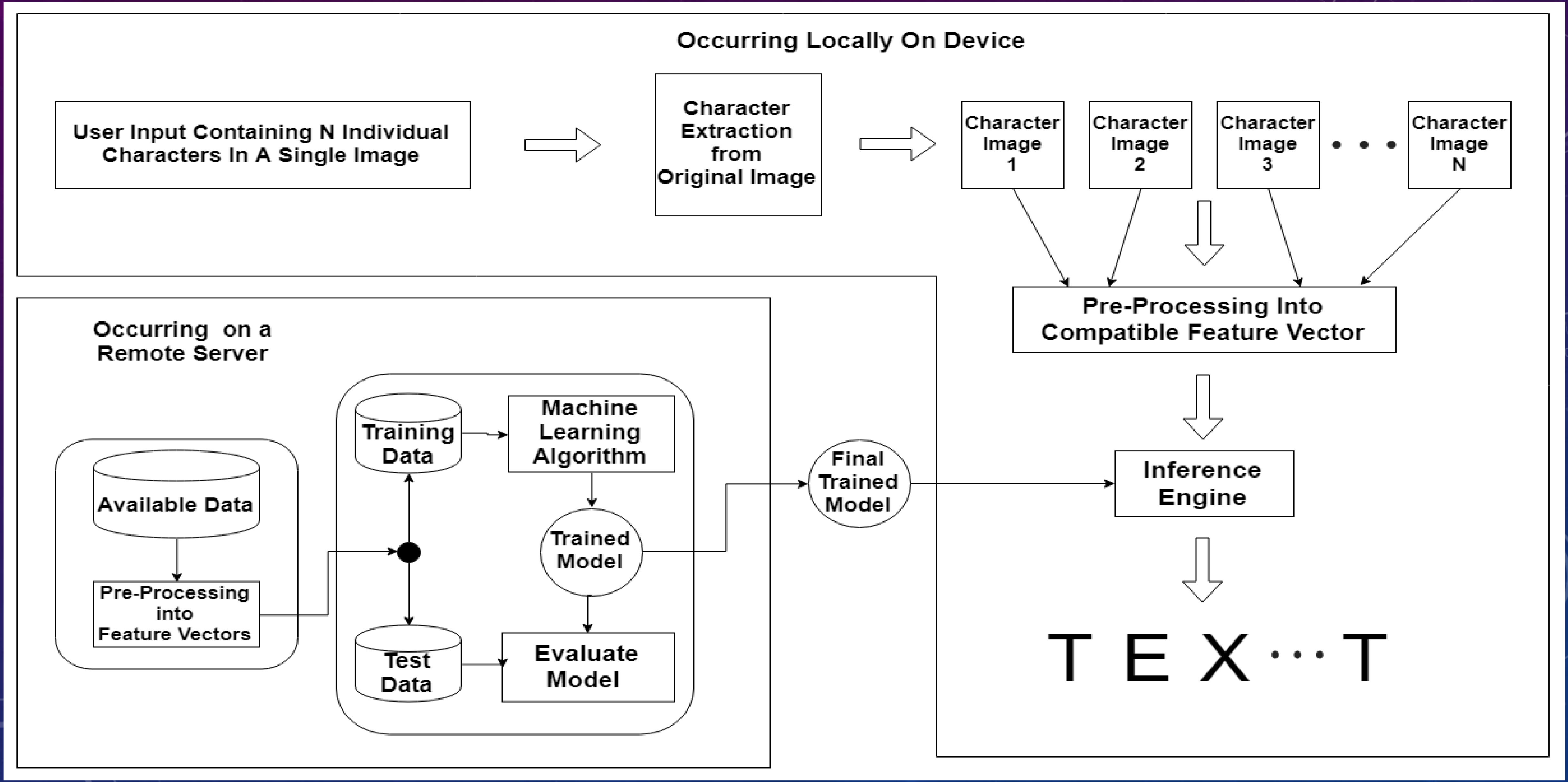
OBJECTIVE AND SCOPE

- The **objective** was the implementation of a system that can:
  - Train a neural network model
  - Deploy it into an Android application running on a smartphone
  - Which reads an image of handwritten text and determines what is written
- The **scope** of the system is:
  - To be able to take an image of handwritten characters as input
  - Recognize what is written
  - Produce the output as text
- **Primary Focus**:
  - **Recognition** of the individual characters once separated from a larger input image
  - **Deployment** of a trained model to a mobile platform (eg: Android smartphone)

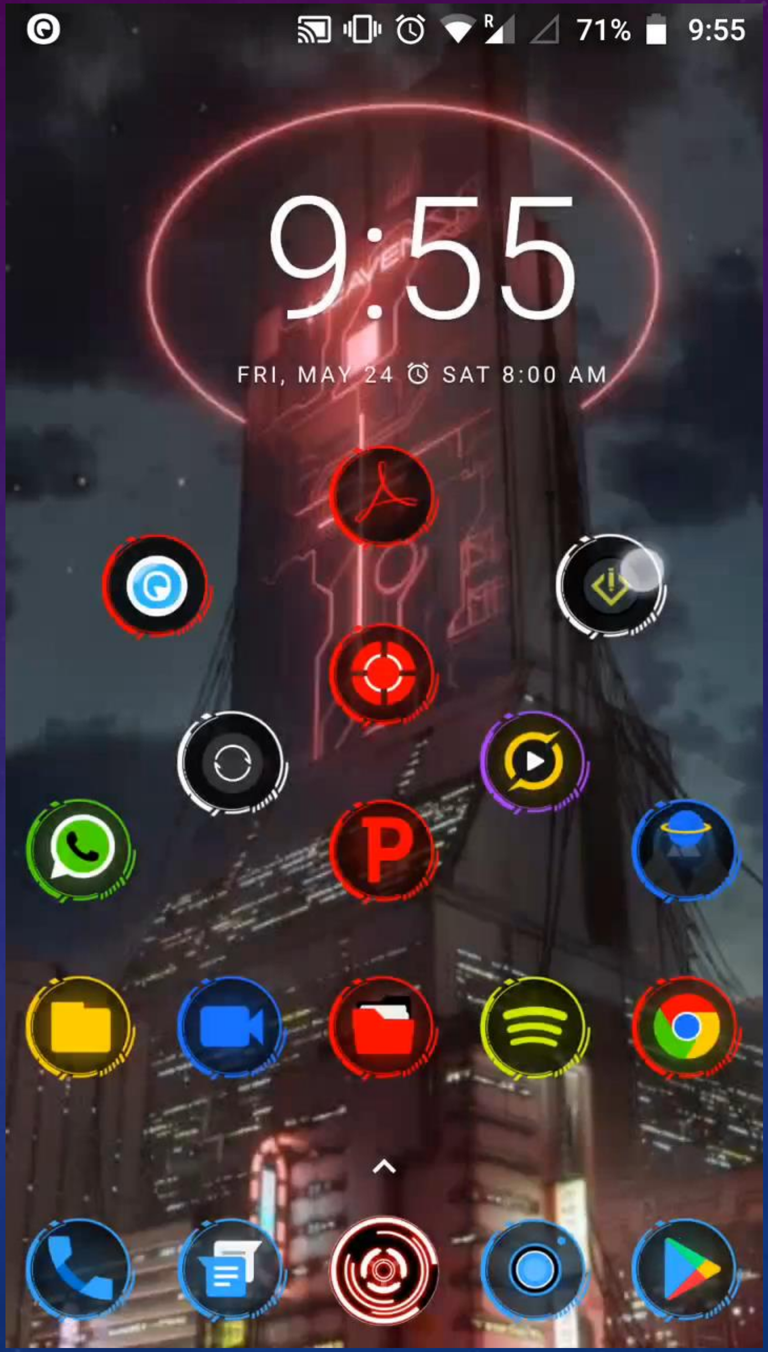
METHODOLOGY

- Handwriting Recognition / Handwritten Character Recognition is inherently a multi-stage process involving:
  - Obtaining the entire image
  - Separating each individual characters in the image
  - Recognizing the characters
- For the recognition to work
  - The images of the single characters need to pre-processed
  - Preprocessed images need to converted into compatible vectors for the inference model
  - The inference model has to be properly trained, i.e. its parameters and architecture properly optimized

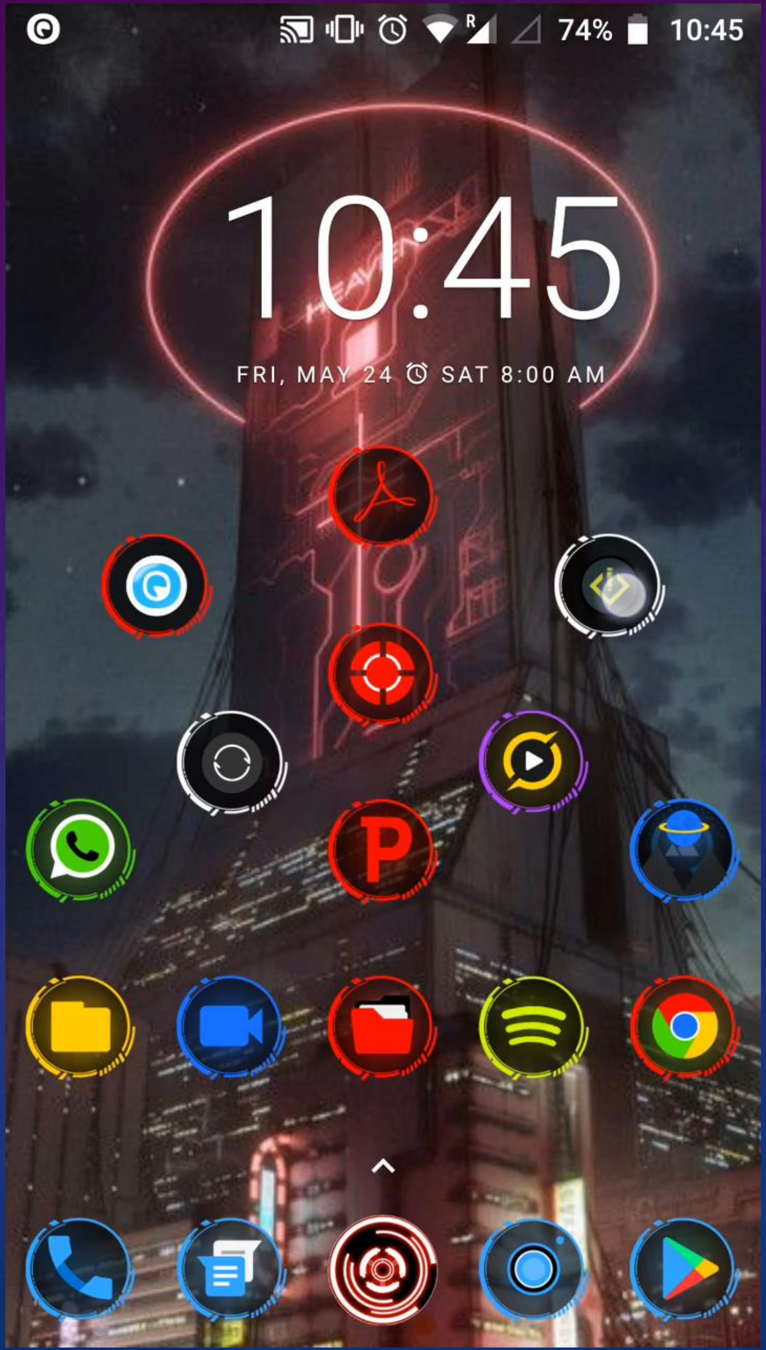
SCHEMATIC DIAGRAM OF THE SYSTEM



DEMONSTRATION:



EXAMPLE USAGE:

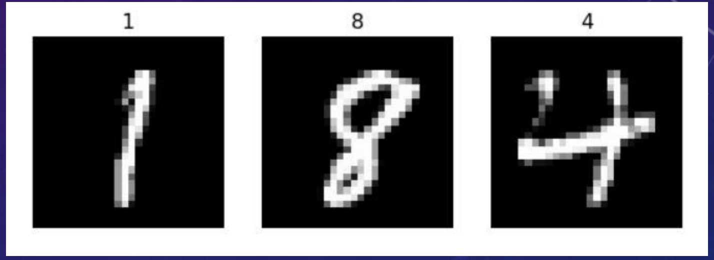


IMPLEMENTATION DETAILS

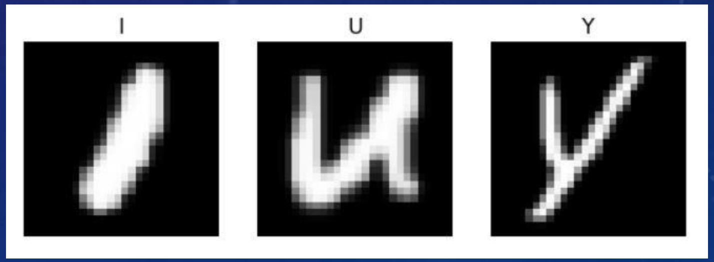
DATASETS:

- **MNIST database of Handwritten Digits:**
  - **Storage Format**: idx-ubyte format
  - **Resolution of each sample**: 28 x 28
  - **Training Samples**: 60,000 grayscale images
  - **Testing Samples**: 10,000 grayscale images
- **EMNIST letter data:**
  - **Storage Format**: idx-ubyte format
  - **Resolution of each sample**: 28 x 28
  - **Training Samples**: 1,24,800 grayscale images
  - **Testing Samples**: 20,800 grayscale images

After proper parsing of the IDX files



MNIST handwritten digits



EMNIST letter data

TRAINING AND TESTING MODELS

- The training process involved multiple steps:
  - Reading the IDX files
  - Generating scaled feature vectors from the read data
  - Configuring and training the Neural Network
  - Evaluating the trained model
- Only after a suitable model has been trained can we proceed to deployment.

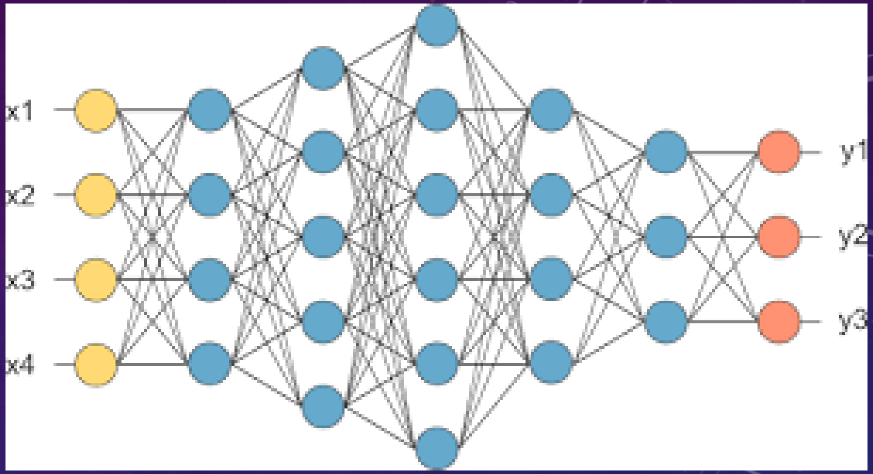
READING THE IDX FILES

- FUNCTION read_idx:
  Reads and interprets an IDX file into a multidimensional array or vector which can be used for computation purposes.
  - INPUT:
    - *filepath*: path to the IDX file
  - OUTPUT:
    - *data*: the array or vector stored in the input file
  - BEGIN:
    1. Open gzip file for reading and extraction of data
    2. *magic_number* = First 4 bytes from file converted it to an integer
    3. *m* = Next 4 bytes from the file converted to an integer
       where, m is the number of samples
    4. IF *magic_number* is 2051:
       i.e. if the file contains images
       1. *nrows* = next 4 bytes converted to an integer
          where *nrows* is the number of rows in the image
          i.e. height of the image in pixels
       2. *ncols* = next 4 bytes converted to an integer
          where *ncols* is the number of columns
          i.e. width of the image in pixels
       3. *shape* = (*m* , *nrows* , *ncols*)
    5. ELSE IF *magic_number* is 2049:
       i.e. if file contains labels
       1. *shape* = -1
    6. ELSE:
       i.e. if file is of an unrecognizable type
       1. show error message
       2. RETURN
    7. *data* = the rest of the bytes that are left to be read
    8. Use *shape* to correctly arrange the read values within *data*
    9. RETURN *data*
  - END

BUILDING AND TRAINING THE MODEL

Sr. No	Units in each Hidden Layer	Accuracy								
		Digits					Alphabets			
		Train Loss	Train Accuracy	Test Loss	Test Accuracy		Train Loss	Train Accuracy	Test Loss	Test Accuracy
1	50	0.0439	0.9861	0.0926	0.9743		0.3772	0.8813	0.4386	0.8691
2	100	0.0206	0.9935	0.0883	0.9747		0.2668	0.9133	0.3742	0.889
3	200	0.0122	0.9959	0.0786	0.9797		0.185	0.9346	0.3616	0.9009
4	500	0.0098	0.9967	0.0938	0.9774		0.1349	0.95	0.3712	0.9059
5	800	0.0125	0.9958	0.072	0.9818		0.1232	0.9538	0.3716	0.9109
6	50, 50	0.0385	0.9876	0.1048	0.9702		0.3212	0.895	0.3853	0.8839
7	100, 50	0.0225	0.9923	0.0987	0.9774		0.2359	0.9201	0.3553	0.8891
8	200, 100	0.0168	0.9947	0.0783	0.9811		0.1615	0.9406	0.3404	0.9045
9	200, 200	0.0178	0.9942	0.1123	0.9768		0.1469	0.9461	0.34	0.9092
10	500, 100	0.0154	0.9951	0.0876	0.9799		0.363	0.906	0.3629	0.9059
11	500, 200	0.0183	0.9942	0.1065	0.9783		0.3639	0.9112	0.3639	0.9111
12	800, 500	0.0213	0.9935	0.0992	0.9792		0.1268	0.9531	0.412	0.9105
13	100, 100, 50	0.0253	0.9919	0.096	0.976		0.3413	0.9237	0.3412	0.8987
14	200, 100, 50	0.0219	0.993	0.0916	0.9772		0.1728	0.9375	0.3203	0.9029
15	500, 200, 100	0.02	0.9941	0.076	0.982		0.1439	0.9471	0.3316	0.9133
16	800, 500, 200	0.0226	0.9935	0.0989	0.9797		0.1428	0.9486	0.3415	0.9149
17	800, 200, 50	0.0202	0.9936	0.0826	0.98		0.1338	0.9502	0.3389	0.9126

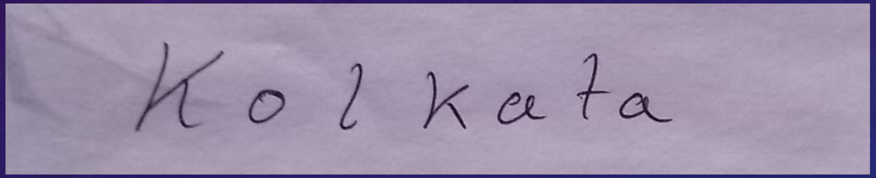
PROPERTIES OF MODEL:



- Both trained models expects an input vector of 784 dimensions (28x28) containing pixel values from the image
- The digits model generates a 10D vector as output containing the probabilities of each class
- The letter model generates a 26D vector containing probabilities
- The model was kept simple, but with relevant accuracy, because the target deployment environment would be a low-power one.

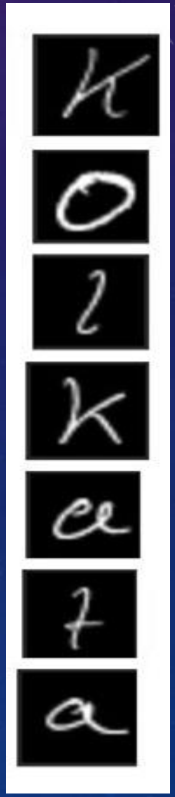
DEPLOYMENT

- Pre-processing the input image:
  - The MNIST and the EMNIST data on which the model was trained was read as pixel intensity values of a grayscale image
  - The image the user will provide as input will have to be read as an 3-channel (BGR) colour image



Original Image

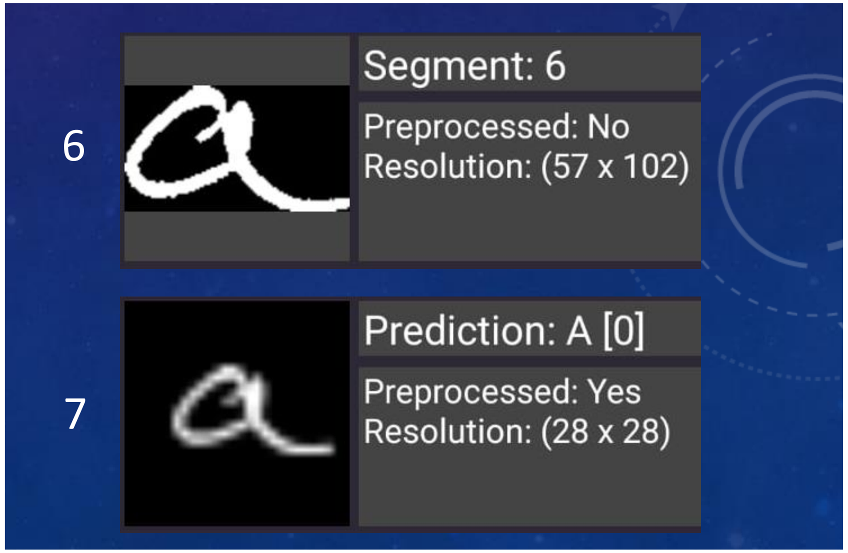
Characters Separated



Preprocessed Separated Characters

PREPROCESSING THE INPUT IMAGE

1. Original Input Image
2. (1) converted to binary image using adaptive thresholding
3. Excess blank spaces removed around (2)
4. Extracted smaller images containing one character each
5. Images in (4) after preprocessing
6. Closer look at single character image
7. Closer look at preprocessed character image



PREPROCESSING STEPS

- Convert RGB image to grayscale and apply adaptive thresholding to get binary image
- Trim excess blank space
- Extract smaller images with one character in each
- Preprocessing each sub-image:
  - Fit the image into a 20x20 box, preserving aspect ratio
  - Pad the fitted image to get a 28x28 image
  - Center the text in the image using center of mass
  - Flatten the image into an 1D array
  - Scale the pixel intensities by dividing the array by 255

PREPROCESSING: RGB TO BINARY

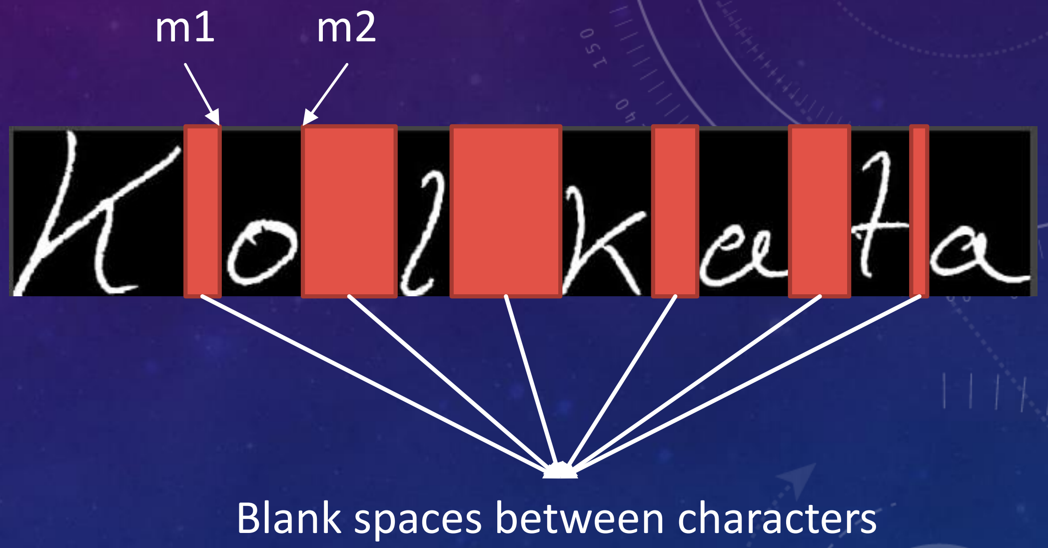
- Convert RGB to grayscale using the following formula:
  - pixel_intensity = $0.299 \times R + 0.587 \times G + 0.144 \times B$
- Use adaptive thresholding on the grayscale image to get binary image:
  - Adaptive threshold computes the value of each pixel by examining its neighbours within a given block-size.
  - The threshold is determined for this local area by statistical operators such as mean, median, etc.
  - Here, adaptive threshold has been used with mean-C which subtracts a constant, C, from the mean of the intensities in the local area and uses it as the threshold
  - This can accommodate change in lighting conditions within the image, like those occurring as a result of a strong illumination gradient or shadows.

PREPROCESSING: TRIMMING BINARY IMAGE

- FUNCTION trim_image:
  - INPUT:
    - *binary_image* : the image obtained after proper adaptive thresholding
    - *pixel_threshold*: no of pixels to ignore while determining region of interest
  - OUTPUT:
    - *trimmed_image*: an image with no blank space surrounding the region of interest
  - BEGIN:
    - *intensity_threshold* = *pixel_threshold* $\times$ 255
    - $r1 = 0$, $r2 =$ height of image -1, $c1 = 0$, $c2 =$ width of image -1
    - WHILE sum of pixel intensities of row $r1$ < *intensity_threshold*
      - $r1 = r1 + 1$
    - WHILE sum of pixel intensities of column c1 < *intensity_threshold*
      - $c1 = c1 + 1$
    - WHILE sum of pixel intensities of row $r2$ < *intensity_threshold*
      - $r2 = r2 - 1$
    - WHILE sum of pixel intensities of column c2 < *intensity_threshold*
      - $c2 = c2 - 1$
    - *trimmed_image* = cropped image between rows $r1$ and $r2$ and columns $c1$ and $c2$
    - RETURN *trimmed_image*
  - END

PREPROCESSING: CHARACTER PARTITIONING

- FUNCTION character_extract:
  - INPUT:
    - *image*: trimmed binary image
  - OUTPUT:
    - *list*: list of extracted character images
  - BEGIN:
    - LOOP over all columns starting from left to right
      - IF both *m1* and *m2* are set to valid column indices
        - Add to *list* region of image between *m1* and *m2* as a character segment
        - Trim the new segment
        - reset *m1* and *m2* to invalid column indices
      - LOOP over all pixels in each column
        - IF any active/white pixel is found
          - flag column
          - break
      - IF column is flagged and *m1* is invalid column index
        - *m1* = current column index
      - ELSE IF column is flagged and *m1* is valid column index
        - *m2* = current column index
    - RETURN *list*
  - END



PREPROCESSING: FIT AND PAD SUB-IMAGE

- FUNCTION fit_image:
  Fits an image of arbitrary resolution into a 20x20 box, preserving the aspect ratio
  - INPUT:
    - *char_image*: image of arbitrary resolution
    - *max_dim*: side of the square to fit into, here 20
  - OUTPUT:
    - *out_image*: the fitted image
  - BEGIN:
    - *r* = rows in original image
    - *c* = columns in original image
    - IF $r > c$
      - $c = \text{round}(c \times max\_dim / r)$
      - r = *max_dim*
      - IF $r > 0$ and $c > 0$
        - Resize image to new fit *r* and *c*
    - ELSE
      - $r = \text{round}(r \times max\_dim / c)$
      - c= *max_dim*
      - IF $r > 0$ and $c > 0$
        - Resize image to new fit *r* and *c*
    - RETURN resized image as *out_image*
  - END

- FUNCTION pad_image:
  Pads an image with blank space to get required dimensions
  - INPUT:
    - *img_in*: a smaller image that needs to be padded
    - *req_r*, *req_c*: the required number of rows and columns
  - OUTPUT:
    - *padded_image*: image with required dimensions
  - BEGIN:
    - *padded_image* = new empty image of *req_r* x *req_c* dimensions
    - *top_row* = ceiling(*req_r* / 2)
    - *left_col* = ceiling(*req_c* / 2)
    - Position *img_in* inside *padded_image* so that the top-left corner of *img_in* aligns with (*top_row*, *left_col*) in *padded_image*
    - RETURN *padded_image*
  - END

PREPROCESSING: CENTER IMAGE

- FUNCTION transform_image:
  Uses outputs from functions like get_center_of_mass and get_transform to center the smaller 20x20 image inside the 28x28 image
  - INPUT:
    - *img*: image to center
    - *shX*, *shY* : translations required along each axis
  - OUTPUT:
    - *centered_image*: image after translation
  - BEGIN:
    - *tr* = [[1, 0, *shX*], [0, 1, *shY*]]
    - apply transform *tr* on *img* and store in *centered_image*
    - RETURN *centered_image*
  - END

- FUNCTION get_transform:
  - INPUT:
    - *img*: an image with an individual character in it
  - OUTPUT:
    - *shX*, *shY*: the translations along each axis required
  - BEGIN:
    - *r*, *c* = rows and columns of *img*
    - *cx*, *cy* = center of mass
      using get_center_of_mass function
    - *shX* = round(*c* / 2 - cx)
    - *shY* = round(*r* / 2 - cy)
    - RETURN *shX*, *shY*
  - END

- FUNCTION get_center_of_mass:
  Computes the center of mass of an image
  - INPUT:
    - *img*: an image
  - OUTPUT:
    - *CoM*: an ordered pair (*cx*, *cy*), indicating the center of mass
  - BEGIN:
    - $rsum = 0, csum = 0, total = 0$
    - LOOP over all rows
      (current row is *ir*)
      - LOOP over all columns in ir^{th} row
        (current column is *ic*)
        - $rsum = rsum + ir \times img[ir][ic]$
        - $csum = csum + ic \times img[ir][ic]$
        - $total = total + img[ir][ic]$
    - $cx = csum\,/total$
    - $cy = rsum/total$
    - $CoM = [cx, cy]$
    - RETURN *CoM*
  - END

PREPROCESSING: FLATTENING AND SCALING FEATURES

Flattening the Image:

1	2	3
4	5	6
7	8	9

⇨

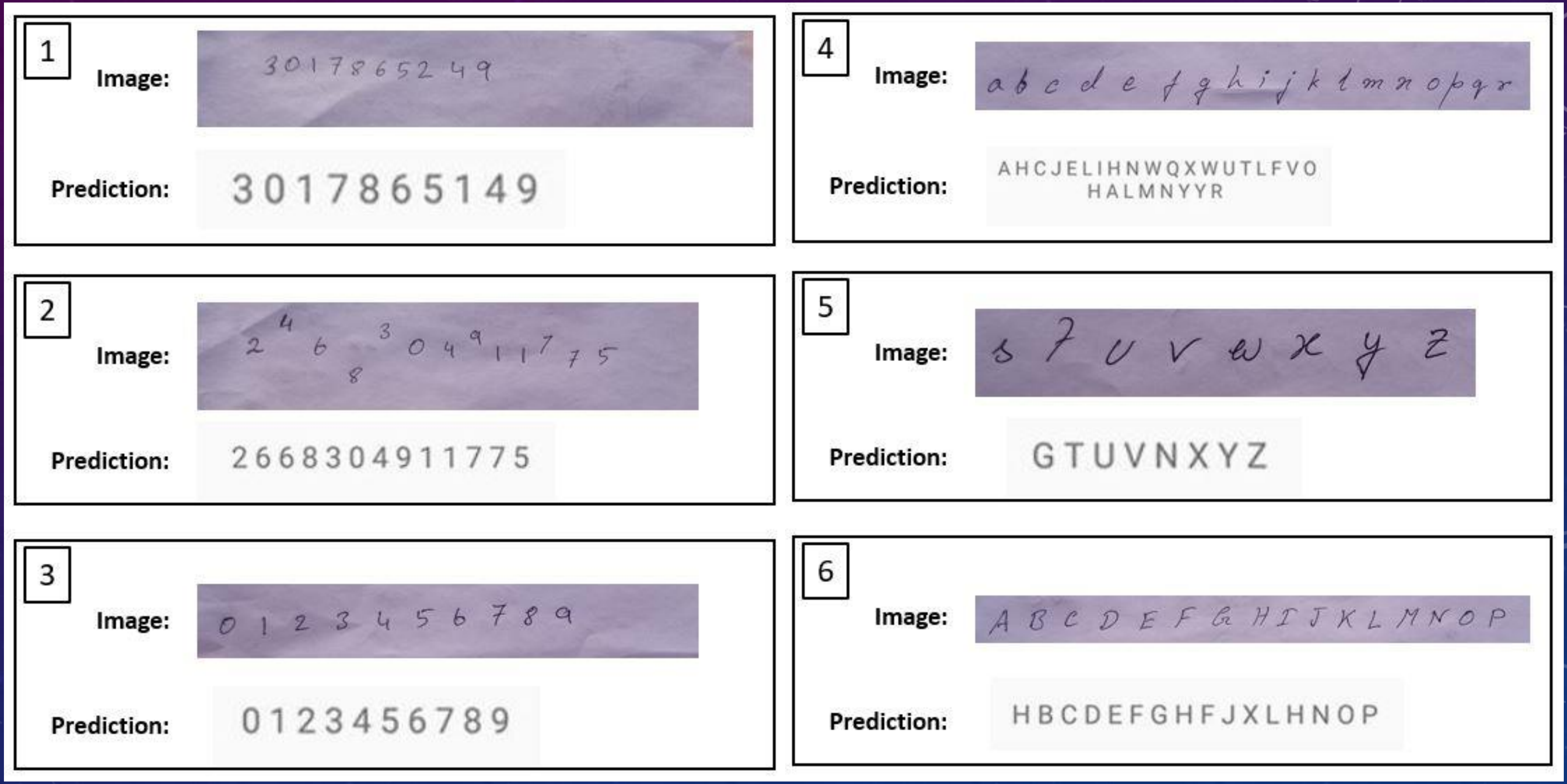
1	2	3	4	5	6	7	8	9

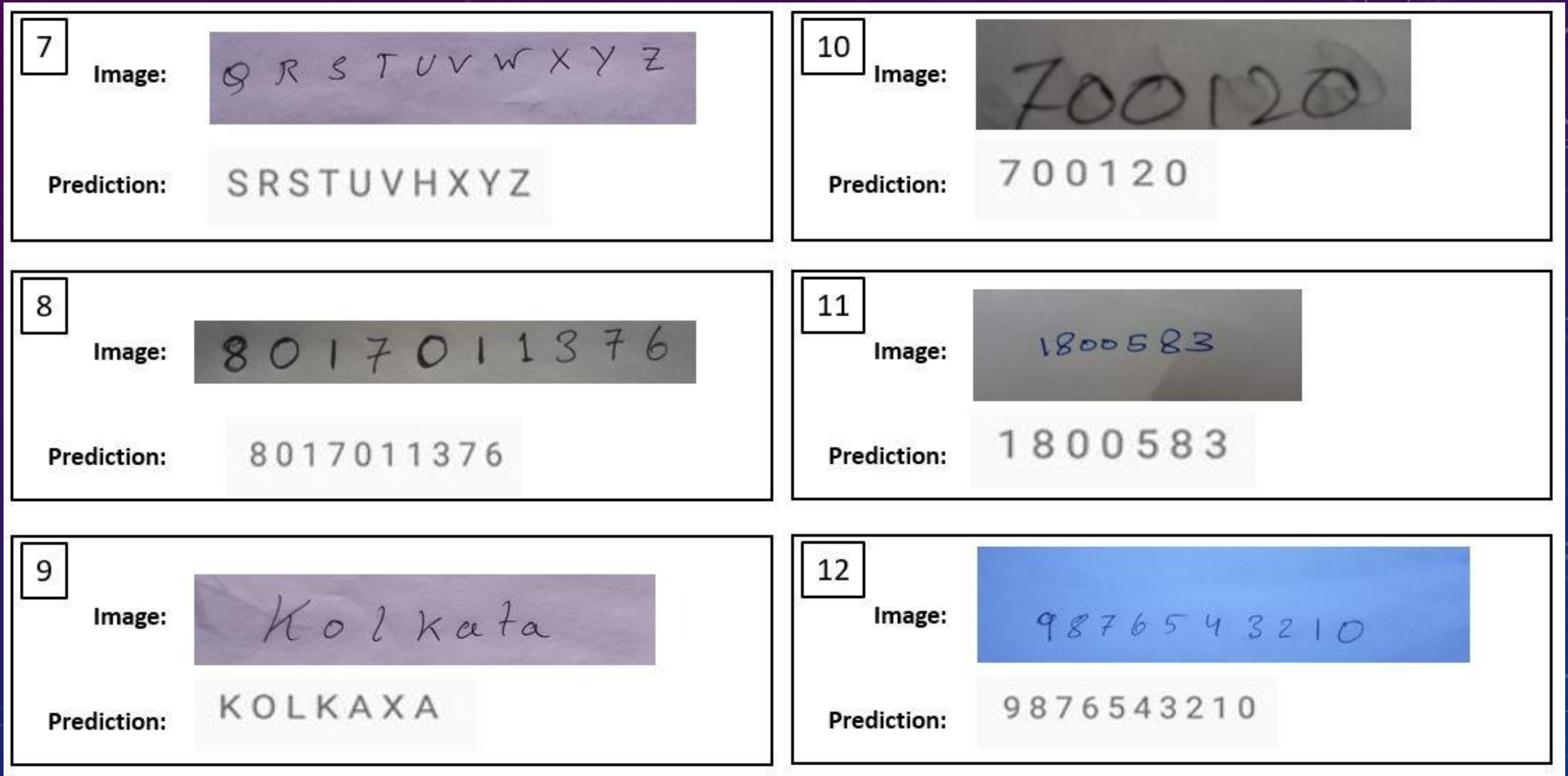
Scaling Features:

1/9	2/9	3/9	4/9	5/9	6/9	7/9	8/9	9/9

In reality, all pixel intensity values are divided by 255.

RESULTS


1
Image:
Prediction:
3017865149
4
Image:
Prediction:
AHCJELIHNWQXWUTLFVO
HALMNYYR
2
Image:
Prediction:
2668304911775
5
Image:
Prediction:
GTUVNXYZ
3
Image:
Prediction:
0123456789
6
Image:
Prediction:
HBCDEFGHFJXLHNOP


7
Image:
Prediction:
SRSTUVHXYZ
10
Image:
Prediction:
700120
8
Image:
Prediction:
8017011376
11
Image:
Prediction:
1800583
9
Image:
Prediction:
KOLKAXA
12
Image:
Prediction:
9876543210

CONCLUSION

- That goal was to be able to implement and deploy a machine learning model that could recognize handwritten characters with an acceptable accuracy on to an Android smartphone via a usable app.
- Understanding the basics of image processing and the popular tools that facilitate handling of images in the Android environment was challenging but also a refreshing learning experience.
- The Android development platform itself is quite vast a development field and over the course of trying multiple solutions to problems and testing debug-builds of the app in the few devices I had available with me, I learned a lot about mobile app development.
- To conclude, getting a working prototype running on an android device with a machine learning model deployed as an inference engine inside was challenging but also rewarding and I am hopeful about the future of this project that began as a major project for my M.Sc. degree.

FUTURE WORK

- **Background Improvements**
  1. Prediction accuracy for letters is not as good as for digits. Improvements can be made by using other datasets for training.
  2. For this project, development was focused on getting the inference engine working once the characters were extracted. In future, the extraction of the characters can be improved, by adding support for cursive handwriting.
  3. Attempts can be made to determine automate the determination of parameters like, block_size, C and pixel_threshold without user intervention. Or use other techniques that do not require these parameters.
  4. Efficiency improvements are always a concern for any software running on mobile devices.
  5. Models can be trained to recognize more languages and number systems apart from digits and the English alphabets.
- **User Interface Improvements**
  1. The current user interface, although usable, could still be improved upon to allow ease of use.
  2. A tutorial can be included in the app itself to inform users on how to use the app
  3. Modern UI design techniques and philosophies can be introduced to allow ease of use. Support for more devices and image formats can be included in future.
  4. Some sort of feedback mechanism could be implemented via which the user can send corrections to misclassified characters

REFERENCES

- [1] I. Transactions, O. N. Pattern, and A. A. N. D. Machine, “The State of the Art in On-Line Handwriting Recognition,” in *The State of the Art in ITS*, vol. 12, no. 8, 2010, pp. 1–6.
- [2] T. Plötz and G. A. Fink, “Markov models for offline handwriting recognition: a survey,” *Int. J. Doc. Anal. Recognit.*, vol. 12, no. 4, pp. 269–298, Dec. 2009.
- [3] L. Xu, A. Krzyżak, and C. Y. Suen, “Methods of Combining Multiple Classifiers and Their Applications to Handwriting Recognition,” *IEEE Trans. Syst. Man Cybern.*, vol. 22, no. 3, pp. 418–435, 1992.
- [4] P. J. Grother and K. K. Hanaoka, “NIST Special Database 19,” 2016.
- [5] C. J. C. B. Yann LeCun, Corinna Cortes, “THE MNIST DATABASE of handwritten digits.” [Online]. Available: http://yann.lecun.com/exdb/mnist/.
- [6] G. Cohen, S. Afshar, J. Tapson, and A. van Schaik, “EMNIST: an extension of MNIST to handwritten letters,” *Proc. Int. Jt. Conf. Neural Networks*, vol. 2017-May, no. February, pp. 2921–2926, Feb. 2017.
- [7] J. Wang, B. Cao, P. Yu, L. Sun, W. Bao, and X. Zhu, “Deep learning towards mobile applications,” in *Proceedings - International Conference on Distributed Computing Systems*, 2018, vol. 2018-July, pp. 1385–1393.
- [8] A. Ignatov *et al.*, “AI Benchmark: Running Deep Neural Networks on Android Smartphones,” in *ECCV*, 2018.
- [9] Python.org, “Python 3.6 Documentation.” .
- [10] Oracle, “Java 8 Documentation.” .
- [11] Google and JetBrains, “Android Studio.” .
- [12] M. Developers, “Jupyter Notebook.” .
- [13] “OpenCV Documentation.” [Online]. Available: https://docs.opencv.org/master/. [Accessed: 22-May-2019].
- [14] Google Brain Team, “TensorFlow 1.13 API Documentation.” .
- [15] Imranhasanhira, “android-file-chooser.” [open source library]
- [16] Department of Artificial Intelligence of University of Edinburgh, “HIPR2 :Adaptive Thresholding.”[Online]. Available: http://homepages.inf.ed.ac.uk/rbf/HIPR2/adpthrsh.htm. [Accessed: 22-May-2019].