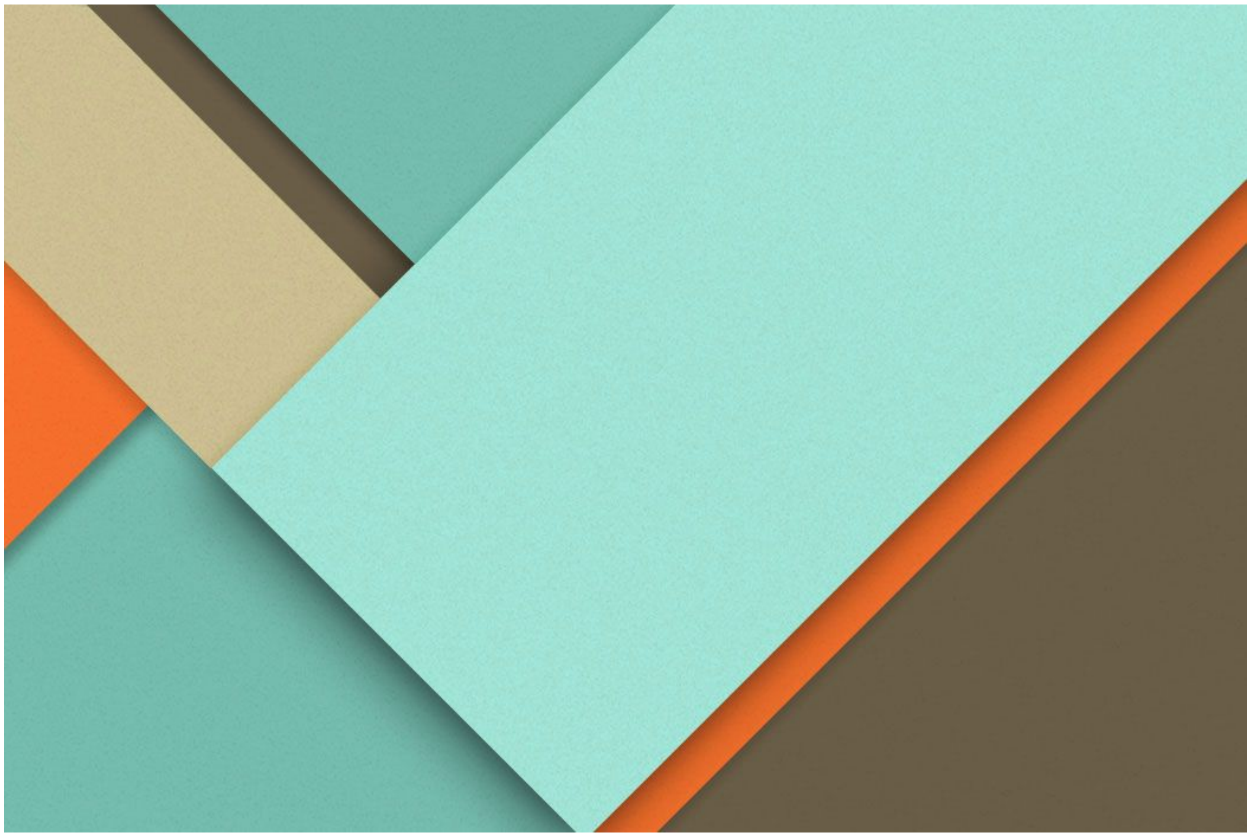


Number Translator

MLND Capstone Project

12.14.2016

Girish Pai
Udacity MLND Student

Definition

I. Project Overview

This project aims to build an Android App, which translates handwritten multi-digit numbers from an image taken from the phone into some other form. As the first prototype of this app submitted for the Nanodegree capstone the following constraints are applied due to resource limitations. All preprocessing and image recognition will be done on the phone. Only the training process will be done offline.

1. The multi digit number will be translated to word form. The same design principles can be applied for translation of numbers from one language to the other.
2. Users are required to take the photo on a light colored background (like whiteboard). This will make the preprocessing steps easier.

Example : (Not an actual App screenshot - image just to get the idea).

The core of the software for this app is a deep-learning model using Convolutional Neural Networks to recognize the individual digits in the image. Preprocessing and segmentation of the digits was done using OpenCV for Android. The model will be trained using publically available MNIST dataset of handwritten digits.

II. Technical Background

A. **Image Representation inside the Computer** :

Images can be thought of as a function,f or I, from R^2 to R :

- f(x,y) gives the intensity or value at position (x,y).

- Defined over a rectangle, with a finite range :
  - [a,b] x [c,d] → [min,max]
- The min,max values indicate range of intensities.
- For a Monochrome (Black and white) images
  - Min = 0 (Black), Max = 1 (White)
- For Grayscale images
  - Min = 0 , Max = 255

Color images can be thought of as three functions stacked together.

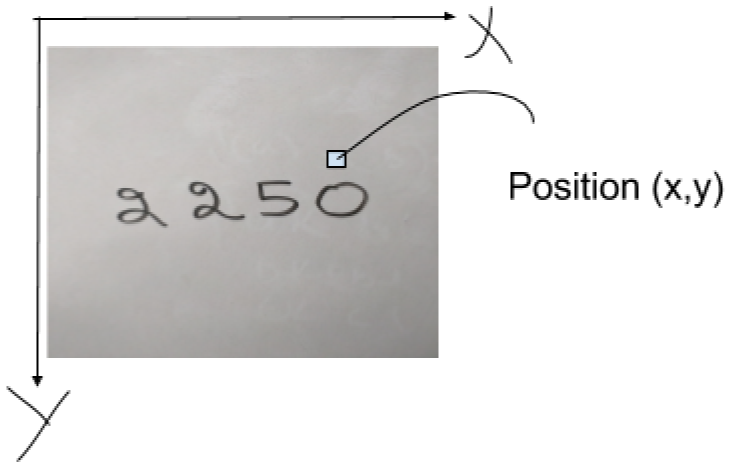
$$f(x,y) = \begin{bmatrix} r(x,y) \\ g(x,y) \\ b(x,y) \end{bmatrix}$$

For each location (x,y) it gives the intensity of the red,green and blue channel.

The range of these intensity values is [0,255].

The below example might make this more concrete.

If the image below is of size 28 * 28 - Range of x = [0,27], y = [0,27].



For convenience the color image is also represented as a 3-dimensional array where in the 3rd dimension has size 3 (for each channel).

B. **Conversion of image from RGB to grayscale** :

In many Computer Vision applications, it is useful to convert the image from color (RGB) to grayscale. This is done by forming a weighted sum of R,G and B components :

0.2989 * R + 0.5870 * G + 0.1140 * B

This results in a single scalar intensity value from [0,255] for every position (x,y)
In the Image matrix as opposed to a tuple of of r,g,b values.

In the case of OpenCV there is a built in function that performs this operation for us.

C. **Noise Filtering** :

Many image processing algorithms work better on images with noise smoothed. This is performed using a filter. There are different kinds of filter, but in this project we used the Gaussian Filter.

D. **Edge Detection** :

Edge detection includes a variety of mathematical methods that aim at identifying points in a digital image at which the image brightness changes sharply. There are different edge detection algorithms. In this project, Canny Edge detector [14] was used in the preprocessing stage.

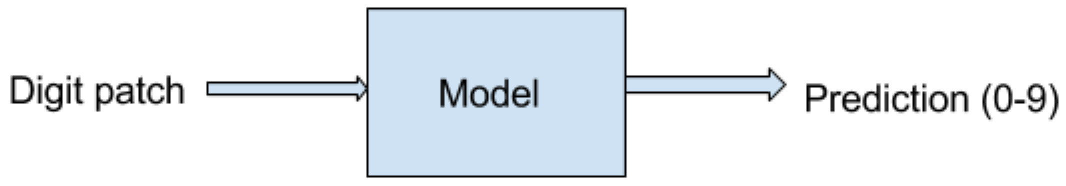
E. **Contour Finding** :

Contour is a curve joining all the continuous points (along the boundary), having same color or intensity. Contours are a useful tool for object detection and recognition. In this project, the objects are the individual digits in the number.

- For better accuracy, use binary images. So before finding contours, apply threshold or canny edge detection.
- In OpenCV, object to be found should be white and background should be black. Hence a preprocessing step is needed to satisfy this requirement.

F. **Digit Recognizer :**

All the concepts discussed above are Image Processing / Computer Vision techniques to be used to segment the digits. Once the individual digit patches are obtained, this is sent to the digit recognizer (which is a machine learning model) to predict its value.



The model used for the prediction will be Convolutional Neural Network (ConvNet).

- A ConvNet architecture is in the simplest case a list of layers that transform the input image volume into an output volume (e.g. holding the class scores)
- There are three main layers - Convolutional, Relu,Pooling and Full Connected.
- Each Layer accepts an input volume in 3-dimensions and transforms it to an output volume in 3 dimensions through a differentiable function.
- Each Layer may or may not have parameters.
- Each Layer may or may not have additional hyperparameters.

G. **Convolutional Layer :**

Convolution is an operation on two functions for a real-valued argument. Let x(t), w(t) be two functions of t. Then the convolution of these two functions (again a function of t) will be given by :

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a).$$

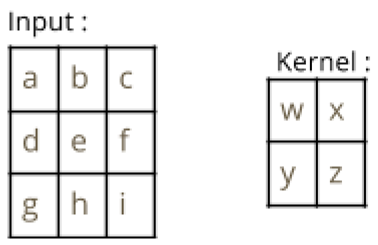
In convolutional network terminology, the first argument (in this example, the function x) to the convolution is known as the input and the second argument (in this example, w) as the kernel. The output is referred to as the feature map.

In machine learning, the input is usually a multidimensional array of data and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm. The functions are assumed to be zero everywhere but the finite set of points. This means that in practice we can implement the infinite summation as summation over finite number of array elements.

Also, more than one axis are convolved at a time. For example, in the case of grayscale image I, which is 2 dimensional, which is our input and a 2-dimensional kernel K :

$$S(i,j) = (I*K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

Example : Consider the following input and kernel (both 2 dimensional). Restricting the output to only positions where the kernel lies entirely within the image. This is known as “valid” convolutions.



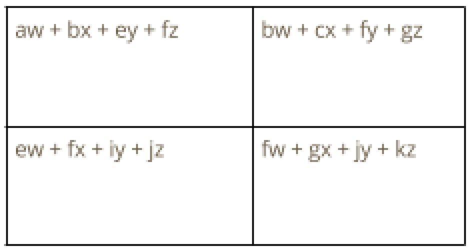
Input :

a	b	c
d	e	f
g	h	i

Kernel :

w	x
y	z

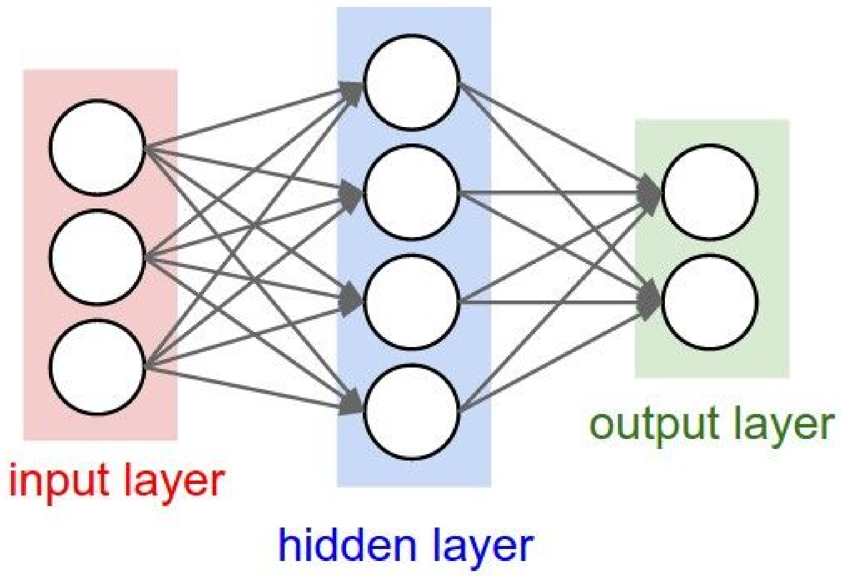
The output is again 2-dimensional with 4 entries.



aw + bx + ey + fz	bw + cx + fy + gz
ew + fx + iy + jz	fw + gx + jy + kz

Following are the two main motivations behind using Convolution in the model :

1. Sparse Connectivity :
   Consider a traditional 2 layer neural network :



This uses matrix multiplies by a matrix of parameters with a separate parameter describing the interaction between each input unit and each output unit. If x is the input, for the neural network :

Out = $W_2 max(0, W_1 x)$

Every output unit interacts with every other input unit as shown above. Clearly, regular neural networks do not scale well to full images.

For example, consider the following :

Input image size : 32 * 32 * 3
Number of weight parameters in 1st unit in hidden layer = 3072

Now let us increase the image size to a realistic one :

Input image size : 640 * 480 * 3
Number of weight parameters in 1st unit in hidden layer = 921600

Moreover, there would be many such units in the hidden layer. Hence the parameters would add up pretty quickly !

Convolutional networks on the other hand have sparse weights. This is because the Kernel is smaller than the input. For example, when processing an image, the input image might have millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels. This means fewer parameters need to be stored, which reduces both the memory requirements of the model and improves the statistical efficiency.

2. Parameter Sharing :

This refers to using the same parameter for more than one function in a model. In a regular neural network, each element of the weight matrix is used exactly once when computing the output of a layer. It is multiplied by one input and never visited again.

In a ConvNet, each member of the kernel is used at every position of the input. This means that rather than learning a separate set of parameters for every location, we learn only one set. The obvious benefit is the less storage. More importantly, this gives rise to equi- variance to translation. For example, when processing images, it is useful to detect edges in the first layer of the convolutional network. The same edges appear everywhere in the image, so it makes sense to share parameters across the whole image.

H. **Non-Linear Activation :**

Each output from the convolving the input image with the kernel is produced by a linear operation. We add a nonlinearity right after this. There are different kinds of nonlinearities which can be used.

- Rectified Linear Unit (ReLu)
- Sigmoid
- Tanh
- Leaky Relu

For this project, I decided to use Relu activation.

I. **Pooling Layer :**

A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. Popular pooling functions are :

- Max Pooling
- L^2 norm of a rectangular neighborhood
- Weighted average based on distance from the central pixel

Pooling helps to make the representation invariant to small translations of the input. Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change. Invariance to local translation can be a very useful property if we care about whether some feature is present than exactly where it is.

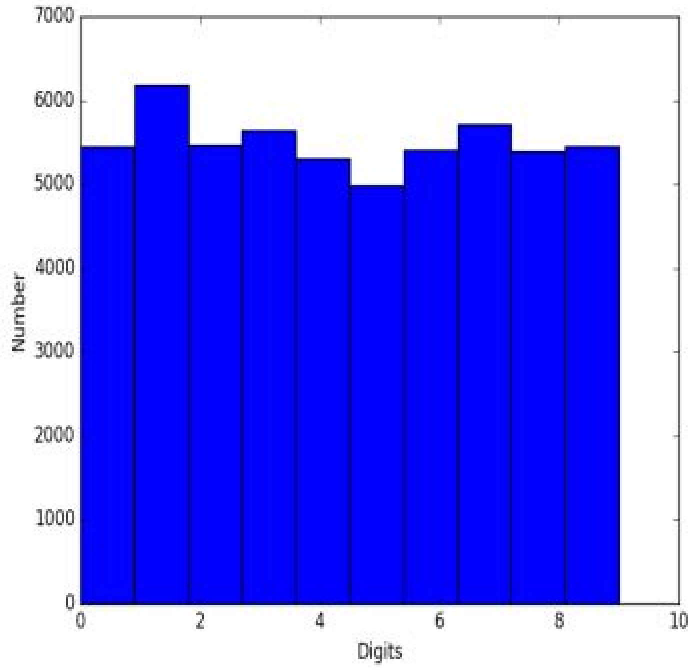
Dataset

For this project, the dataset used was the publically available MNIST dataset of handwritten digits [15].

I. Exploration

Total number of samples in training dataset = 55,000.

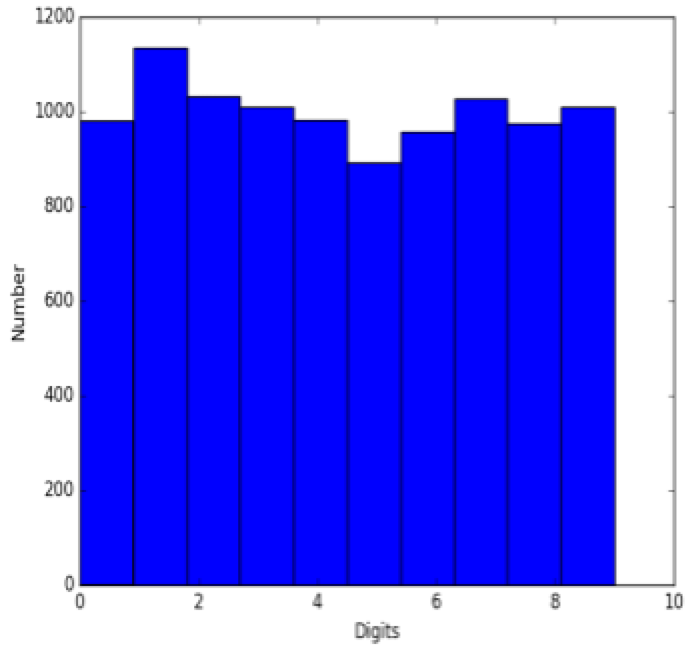
The per digit label breakup is given in the histogram below.



As can be seen, there is a slight imbalance in the training dataset. The range is roughly [5000,6000].

Total number of samples in the test dataset = 10000.

Per digit breakup given by histogram below :



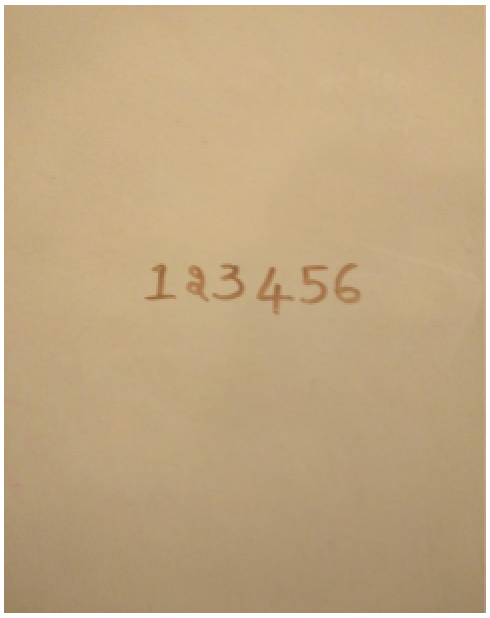
Machine Learning Pipeline

The ideas for pre-processing and segmentation were obtained from [1].

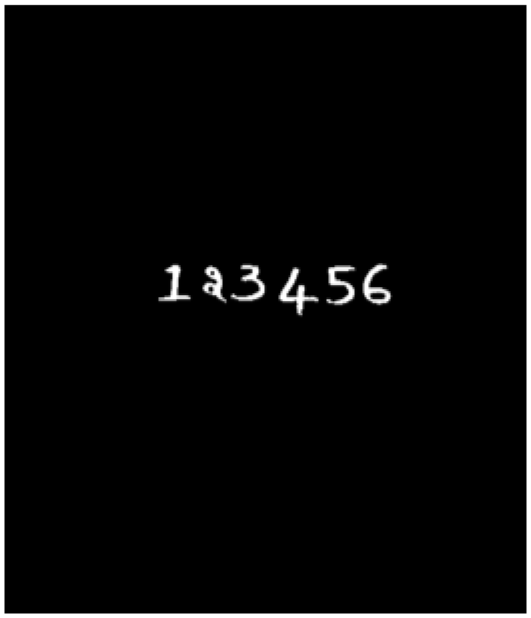
I. Preprocessing

Any photo taken from the phone can have a lot of noise mainly in the form of shadows, which makes the segmentation task more difficult. To filter out this noise and amplify our region of interest (the digits in this case) we perform the following steps.

- Convert image to grayscale.
- Use Canny Edge detection to locate the digits, amplify the digits by making it completely white (pixel value = 255), and make the background black (pixel value = 0).



Original Image



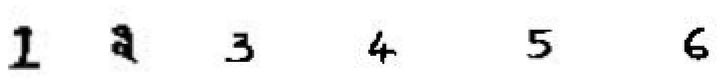
Preprocessed to make ROI white and background black

II. Segmentation

Digit patches from the multi-digit number are obtained with the following steps :

- Find contours of the pre-processed images.
- Find bounding rectangles of all the contours.
- Each digit will have a unique bounding rectangle.

- Since the multi-digit number is in English and should be read from left to right, the bounding rectangles are sorted in the order of the X coordinate.
- Each digit patch is rescaled to size 28 x 28 which is then used as an input to the Convolutional Neural Network.


Segmented digits to be sent to the recognizer.

III. Recognition

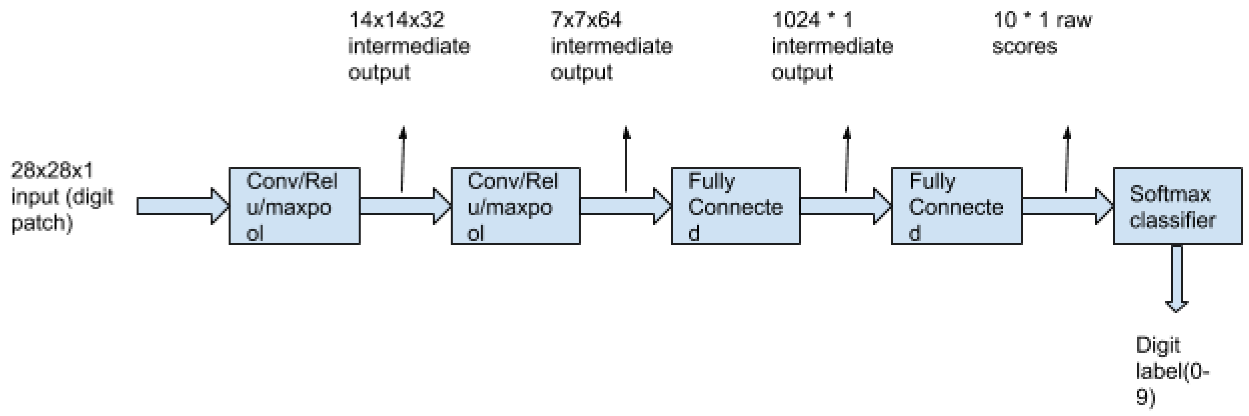
This is the step where each individual digit patch is recognized by the Convolutional Neural Network. The model is trained offline using the MNIST handwritten dataset.

Implementation

This is split into two sections :

1. Training the model (ConvNet) to recognize handwritten digits (used MNIST dataset).
2. Android application development with OpenCV.

Training was done on publically available MNIST dataset of handwritten digits using Tensorflow library in Python. Used a custom Convolutional Neural Network with 2 Conv layers followed by 2 fully connected layers.



Training was done using Stochastic Gradient Descent (SGD) [10] using Adam optimizer [12], which has faster convergence than vanilla SGD. Gradients for the update were computed using backpropagation algorithm. Dropout [13] was used for regularization to prevent overfitting.

Once the model was trained, the graph was saved with the trained parameters for exporting it to the Android Application.

Android Application development was done using Android Studio. Tensorflow model was incorporated into it using Java Native Interface. The idea for that was borrowed from [1] and [2].

Following is the code structure of the app.

Java :

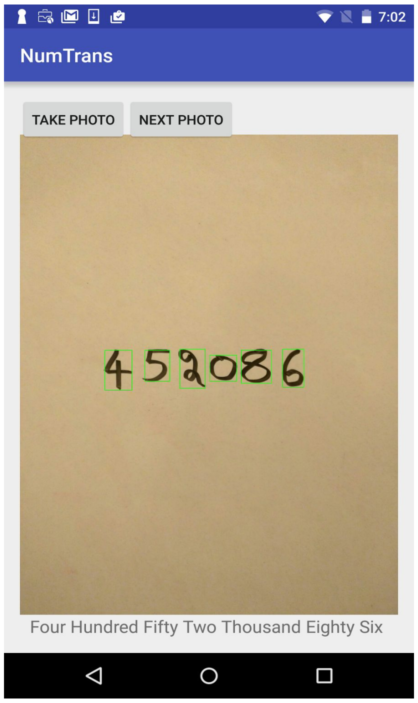
- MainActivity.java : This is the entry class for the app. Sets up the basic interface with the camera and the buttons. Calls the methods from the remaining classes to make the app working.
- IOUtils.java : Utils class for image file handling tasks.
- ImageProcessor.java : Class containing all the image processing functions.
- DigitDetector.java : Class containing the JNI methods which eventually use the tensorflow model to perform the digit recognition. The JNI implementation of the methods was borrowed from [2] as is to reduce development time. The source code of JNI implementation is available on Github for completeness.
- NumToWordConv.java : Class containing methods to convert number to word form.

XML :

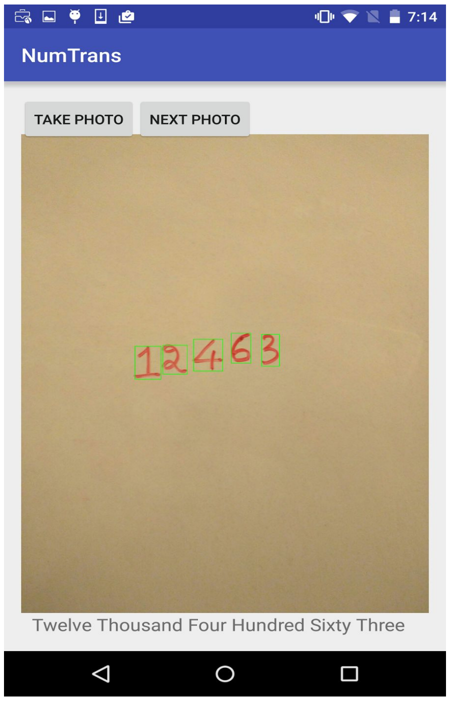
- Activity_main.xml : File determining the app layout.

Results

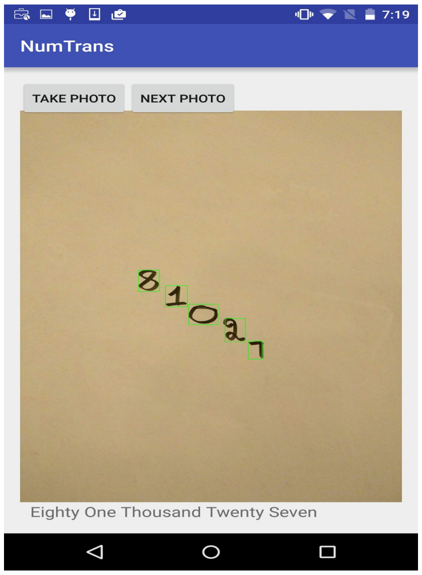
1. Model Training : The custom ConvNet was able to achieve a test accuracy of 99.17% which is good.
2. Android App : The biggest challenge was to get the app working to recognize multi digit numbers. The app was run on Google Nexus P device running Android version N. Below are few of the sample photos taken :



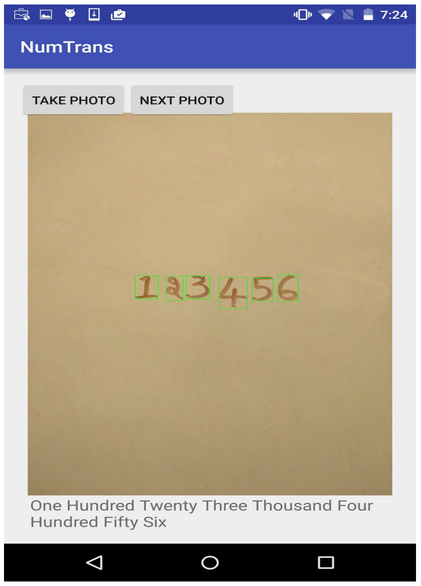
Four Hundred Fifty Two Thousand Eighty Six



Twelve Thousand Four Hundred Sixty Three



Eighty One Thousand Twenty Seven



One Hundred Twenty Three Thousand Four Hundred Fifty Six

Following are the observations based on the sample of photos taken on a clean whiteboard :

- The app performs really well when the contrast is highest - when we use black marker for example.
- The app performs really well even when there are some shadows falling on the whiteboard.
- Numbers written both horizontally and vertically are correctly recognized. The only caveat is it deciphers the number from left to right like is common in the English language.
- Different colors were tried - black,red,brown,green, for all of them the recognition was spot on.
- Performance is deteriorated when the lighting is not that good.

3. The Nexus 6P device had got a lot of memory (64 Gb) and hence space was never an issue though I was dealing with high quality bitmaps taken from the camera app. **Upon testing on a device with lesser memory (Samsung S5 - 16 GB)**, the same app crashed almost immediately after taking the photo due to insufficient memory. This was rectified by rescaling the bitmap to a manageable level.

Conclusion

The NumTrans Adroid App successfully recognizes multi-digit handwritten numbers written on a whiteboard. It feels great to see my model deployed in an actual product and seeing it in action.

This app was not tested on other surfaces due to lack of time. Also, this being a Machine Learning project and not an Android Development one, I decided to not worry about gracefully exiting the app incase of errors.

There are two ways to improve this app.

- Improve the image processing and make the segmentation step better. The bottleneck right now is splitting the number into individual digits. If that is done correctly, the model to recognize individual digit has been trained almost close to human level accuracy. This requires extensive Computer Vision knowledge. Right now the app is not robust to bad lighting / noise.
- The other approach could be to try to make a more complicated deep model, which takes the raw image (with minimal pre-processing) and figures out everything on its own (Localization,Segmentation etc). Something on these lines is done in [4], but it requires a lot of compute power to train such a deep model. Even with the GPUs available to them the authors of that paper took weeks to train the model. This was beyond my budget.

There were lot of challenges faced in this project right from the onset.

- Getting Tensorflow model to port to Android Studio was one of the hardest parts - had to search a lot on the web and figure out from multiple sources.
- Making OpenCV work with Android Studio turned out to be hard due to limited information available. Thankfully a demonstration video on Youtube [9] was the savior.
- Since I was new to Android, there was a steep learning curve for me to develop the app. Having prior knowledge of Object Oriented Programming helped.
- But to finally see the App fully functional was very rewarding and worth all the effort. Not only did I understand Deep Learning, but also gained exposure to Computer Vision and Android programming.

Reference

[1] http://web.stanford.edu/class/cs231m/projects/final-report-yang-pu.pdf

[2] https://www.tensorflow.org/

[3]https://github.com/miyosuda/TensorFlowAndroidMNIST

[4]https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/42241.pdf

[5] http://opencv.org/

[6] http://www.deeplearningbook.org/

[7] http://cs231n.github.io/

[8] https://www.udacity.com/

[9] https://www.youtube.com/watch?v=nv4MEliij14&t=21s

[10] https://en.wikipedia.org/wiki/Stochastic_gradient_descent

[11] https://en.wikipedia.org/wiki/Backpropagation

[12] https://arxiv.org/abs/1412.6980

[13] https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf

[14] https://en.wikipedia.org/wiki/Canny_edge_detector

[15] http://yann.lecun.com/exdb/mnist/