

Project Report(ITSP 2018)

Handwriting and Text Recognition

Drishti
Team Leader-Anuj Diwan

July 9, 2018

1 Team

1. Anuj Diwan - 170070005
2. Vijaykrishna G - 170050090
3. Pranay Reddy S - 170070022
4. Rahul Bhardwaj - 170050012

2 Software used

1. Octave (based on MATLAB)
2. TensorFlow on Google Colab with GPU
3. Paint
4. L^AT_EX for writing our report (this!)

3 Introduction

We first wanted to develop our algorithm for a relatively noise-free image consisting of several horizontally aligned lines. We also wanted to learn exactly how each step in our algorithm functions. Three steps were required:

1. Line Segmentation
2. Given a line, segmentation into:
 - (a) characters and
 - (b) words
3. Given a character, recognition of the character using a neural network of suitable architecture after necessary preprocessing .
4. Link to demo on YouTube: <https://youtu.be/-5k1hb-mHK8>

4 Line Segmentation

1. We first converted the image from coloured to grayscale to black-and-white using `rgb2gray` and `im2bw`.
2. Then we identified the first line of the document by finding how many black pixels were there in each row of pixels of the image and calling it 'whitespace' if less than c_{line} pixels were black, where we chose $c_{line} = 0$. The first such white row is the last y-coordinate of the first line. Using a loop, subsequently cutting out each first line and identifying the next line as the first line, we identified all lines.
3. `im2bw` calculates a global threshold for binarization (conversion to black and white). Because of actual varying local thresholds, each character or word is not correctly binarized, resulting in combining of different characters into one or splitting of one character into two, as can be seen in the images below. Thus, we had the important insight of using `im2bw` repeatedly as a local threshold. Thus, after each line is cut, we apply `im2bw` on the (cut) grayscale source image. This gave better results.



Figure 1: 4 characters combined into one

5 Character and Word Segmentation

1. Given a horizontal line, we segment it into characters. Each column which has all its pixels white is classified as whitespace. We called each whitespace-delimited area a 'character'.
2. We then find contiguous whitespaces. We denote the maximum contiguous whitespace length, w_{max} and denote the threshold for word spacing vs. character spacing as

$$w_{th} = c_{word} \times w_{max}$$

where we chose $c_{word} = 0.5$ as it seemed to work well in general. If the length of a given whitespace w is $\geq w_{th}$, it separates two words, else, it separates two characters.

3. Then, given each word, we reapply our character segmentation (from point ??), by reapplying `im2bw` to only the (grayscale) word. This binarizes using only the word, effectively local binarization, that helps with the issues mentioned in point ?? of the previous section.
4. Now, our final list of characters and words has been obtained. We now calculate the topmost and bottommost y-coordinates for each character. Each character's bounding box has now been calculated.

6 Character Recognition using Neural Network

1. We developed a multi-layer neural network training algorithm from scratch on Octave. We then trained it on 28×28 pixels-sized images of letters, from the EMNIST database at <https://www.nist.gov/itl/iad/image-group/emnist-dataset>. Although originally the code was mostly vectorized, an additional non-trivial vectorization sped up things so that we could train

with reasonable speed (≈ 2 hrs) on our local CPU. We gradually went from 40% to 86% test accuracy trying different sizes and regularization parameters.

2. We reached a maximum of 86% accuracy on the test set using Octave on our local CPU and the Google Compute engine. Therefore, we trained our network on TensorFlow (on Python) with the Google Colab GPU resulting in a higher accuracy of 92% on test set and a much faster training time (≈ 10 min).
3. In many cases, overfitting is observed which indicates that the model complexity (architecture) was sufficient but it didn't generalize well. Fine tuning regularization played an important role in the generalization of the trained weights. Also due to the sheer size and computational requirements of the training, TensorFlow with GPU was an absolute necessity to get much quicker training times.

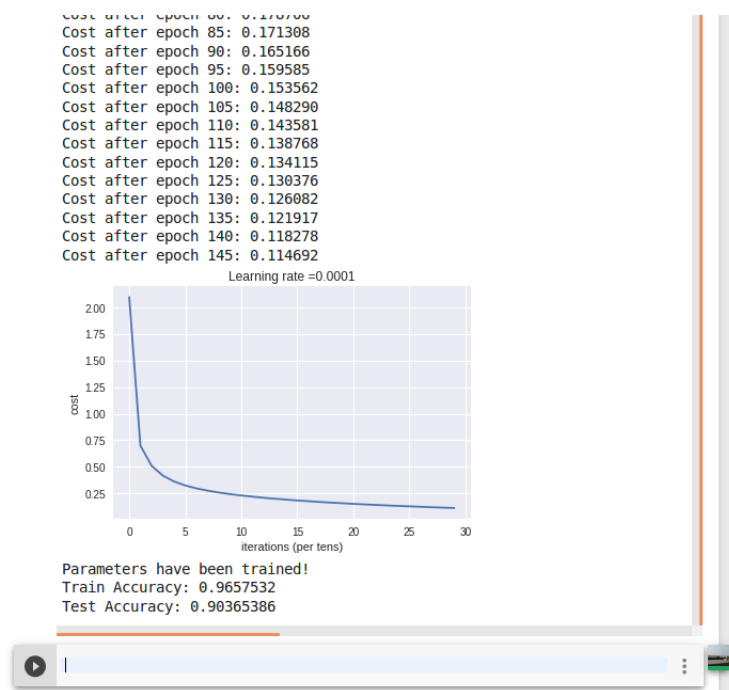


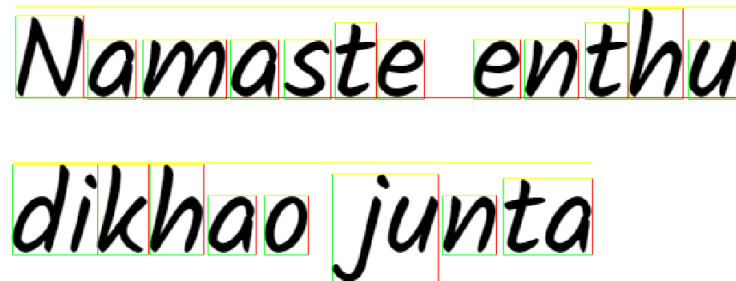
Figure 2: Training on Google Colab Tensorflow

6.1 Neural Network Architecture and Hyperparameters

1. We are currently using a two hidden-layer neural network with hidden layer sizes 1400 and 700. We used mini-batch gradient descent with a mini-batch size of 512 in order to speed up the optimization algorithm. We used Adams Optimization Algorithm which is an advanced optimization algorithm compared to simple gradient descent available on TensorFlow. The flow is shown below:

$$INPUT \rightarrow [Z1 \xrightarrow{\text{RELU}} A1] \rightarrow [Z2 \xrightarrow{\text{RELU}} A2] \rightarrow [Z3 \xrightarrow{\text{SOFTMAX}} P]$$

2. L2 Regularization also played an important role in increasing the test accuracy. We found $\lambda = 0.1$ to be a good choice. The number of epochs in the mini-batch gradient descent was set to 150. With these settings, we obtained a train accuracy of ≈ 96 percent and a test accuracy of ≈ 92 percent. Dropout regularization improved the test accuracy to 93%.



Namaste enthu
dikhao junta

```
Command Window
>> fullpredict
namaste enthu
dikhaojunt
>>
```

Figure 3: Character Segmentation and Recognition

WEAREMACHAXX
DRISHTI
ANUJ
PRANAY
VIJAY
RAHUL

```
>> Paragraph
>> fullpredict
p f a n a y m a c n a y a
>> Paragraph
>> fullpredict
w e a r e m a c h a x x
u r l s h f l
a n u j
p r a n a y
v l j a y
r a h u l
>>
```

Figure 4: Another prediction

7 It All Comes Together

1. We take the characters recognised by the character segmentation, scale them to 28 pixels vertically and then apply white-space padding on either side to make it 28×28 . Next we apply a Gaussian filter with $\sigma = 1$ just like our training data. Below are some examples.
2. Finally, we recognised that while the problem of splitting of one character into two has been solved by local threshold binarization, the problem of clubbing of multiple characters into one character still remains. Because these characters were connected even in the source image, no amount of image processing can effectively separate them. Thus we wrote a completely original piece of code using dynamic programming that solved this problem, as explained in section ?? .

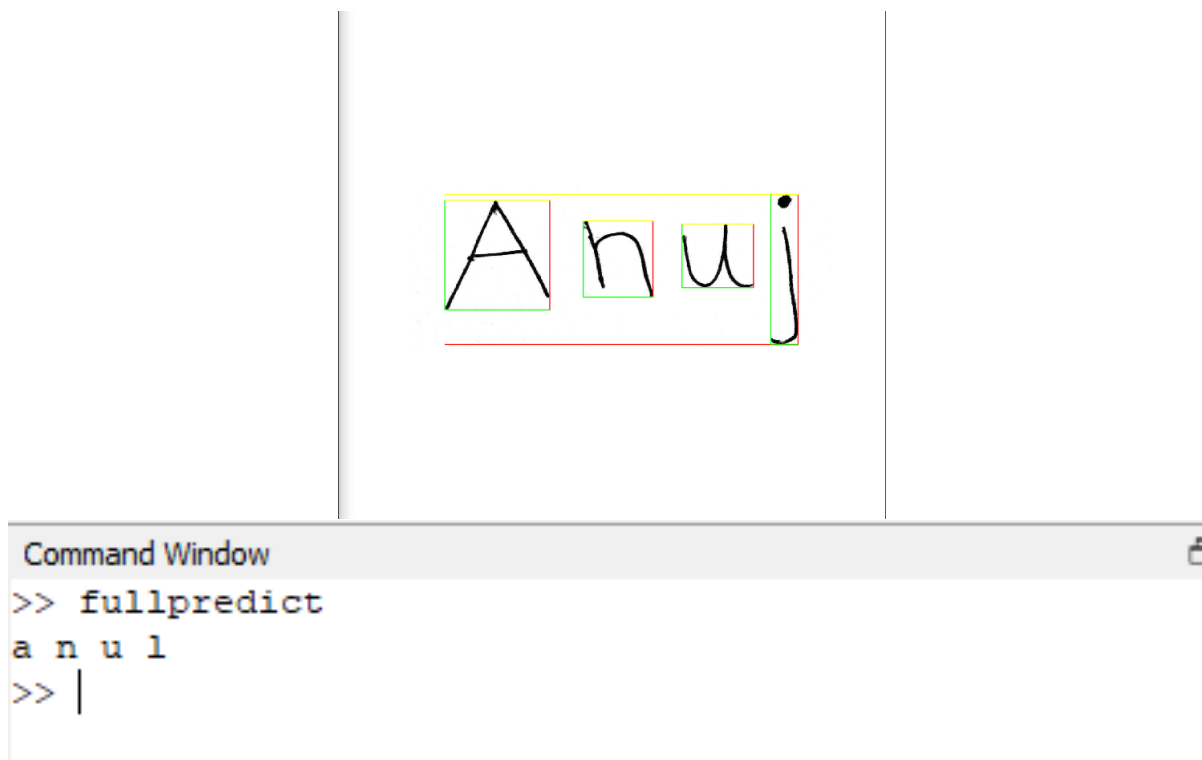


Figure 5: Actual handwriting. The 'j' gets recognised as 'l' because the bend for 'j' isn't as prominent as it is in the training set.

8 Character Segmentation using the Neural Network

1. Our code, `predictcombined.m`, splits up the characters from a 'combined-character' image given the number of characters to split into. This is done by trying all possible combinations of splitting the image up (using dynamic programming), feeding it to our trained neural network and choosing the one that gives the highest overall prediction confidence.
2. After some thought and after reading a paper about confidences in neural networks, we realised that while the SOFTMAX output gives the correct prediction, the value of the SOFTMAX output does not precisely equal the prediction confidence, and in general, more advanced techniques are required to extract out the confidence of a prediction given by a neural network.
3. We however decided to continue using the SOFTMAX output as the prediction confidence as it was giving reasonably good results.
4. While doing character segmentation from section ??, we maintained a quantity l_{av} denoting the average length of a character over all the characters. For each 'character', if $l \geq c_{joined} \times l_{av}$, where we chose $c_{joined} = 1.4$, then it contains a 'combined' character. We then run `predictcombined.m` with the number of combined characters as 2 or 3 and take the output which gives the maximum confidence.
5. Some results are shown below:

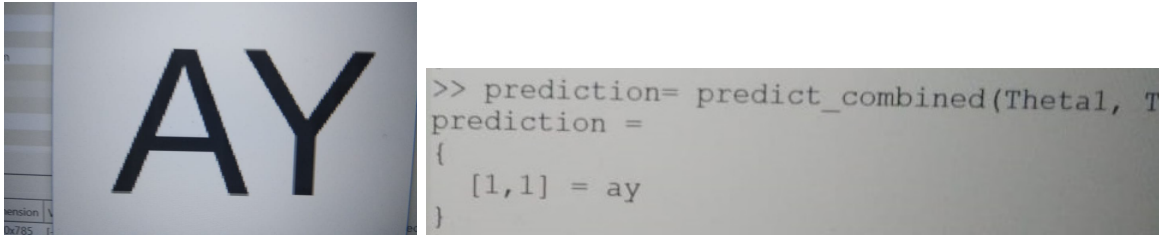


Figure 6: Predictions using the DP Algorithm

9 The Bottleneck

1. Despite all these optimizations, our output was not as good as we hoped. Because of this, we decided to run our algorithms only on digits(much simpler to train on) rather than characters to see where the bottleneck was. We got a train accuracy of 99.9% and test accuracy of 99%
2. We got great results using only digits and our original character segmentation algorithms. Thus, we believe the bottleneck was in the character trained model(which was the best model possible using a normal neural network) and we read on different types of models like bidirectional LSTMs. We didn't fully learn and implement this as it was a bit advanced.

170050090
160040091
130060665
140090052
150060069
130080038

```
>> fullpredict
1 70 08 0 90
1 6 0 0 9 0 0 9 1
1 3 0 0 6 0 6 6 5
1 9 0 0 90 05
1 8 0 6 0 0 6 9
1 3 0 0 80 03 8
>> |
```

Figure 7: On Digits

10 Conclusion

1. This completes our ITSP project. Since state-of-the-art OCR is much more advanced than what we have made here, there are always places to improve upon. Some things we identified include a better model for character identification, removal of different types of noise *without* affecting the characters.
2. We tried ideas like Bernsen binarization and Principal Component Analysis to speed up the learning algorithm but discarded these in favour of other, better ideas.

11 References

1. TensorFlow Tutorial of Andrew NG's Deep Learning Course on Coursera:
<https://www.coursera.org/learn/deep-neural-network/home/welcome>
2. Andrew NG's Machine Learning Course on Coursera:
<https://www.coursera.org/learn/machine-learning/home>
3. A few ideas from <http://neuralnetworksanddeeplearning.com/>
4. Implementation of Bernsen binarization from
<https://tinyurl.com/BernsenBinarization>
5. Octave functions from <https://wiki.octave.org/Octave-Forge> and
<https://www.gnu.org/software/octave/>
6. Computer Vision course on Udacity:
<https://in.udacity.com/course/introduction-to-computer-vision--ud810>