Optical Character Recognition Pipeline for Image of Text to Text File Conversion

Andrej Coleski, Angad Sandhu, Pramat Batra, Zachary Apell
University of Michigan EECS 442: Computer Vision
{candrej, angsan, pramat, zapell}@umich.edu

Abstract

Our goal in this paper is to propose an optical character recognition (OCR) pipeline that can correctly read an image only PDF file utilizing a Convolutional Neural Network (CNN) trained on a computer alphanumeric character dataset. Our proposed framework is essentially a three part pipeline. We needed to read the PDF, build a model, and predict the PDF text. The first part of the pipeline involves character recognition. We achieved this by developing an algorithm that drew bounding boxes around individual characters in PDF files. For the next part of the pipeline we built a CNN model and trained and validated it on the alphanumeric dataset. To complete the pipeline, the model was used to predict the text of the PDF and the results were compared. Our framework is capable of identifying characters of many different fonts successfully. The proposed method shows performance of different models and different fonts.

1. Introduction

This report describes the detailed methodology used to carry out text recognition in PDFs with computer generated text. Our model takes text from PDFs and converts it into ascii text for english alphanumeric characters. It learns these alphanumeric characters from a dataset and uses it to predict the text in PDFs. The report describes the Optical character recognition pipeline, which includes the bounding box, convolutional neural networks and preprocessing techniques, used in our project.

Optical Character Recognition (OCR) is the use of technology to distinguish printed and handwritten text characters inside digital images of physical documents [1]. It is a technique that utilizes deep learning to predict text in images. Over the past years, OCR frameworks have evolved a lot, but not to the extent that where they could be 100% for any image size or quality. [2]

1.1. Problem and Alternative Approaches

Our project solves the problem of converting PDF text with computer generated text to alphanumeric ascii characters.

Optical character recognition is a popular technique used to detect printed and handwritten text characters in images. Over the past few years OCR has been gaining popularity and being able to identify what is present in the image opens up a new horizon of opportunities [2].

There are many softwares/APIs available that do a pretty good job of processing an image.

- Google Vision API: It is one of the most popular APIs available for image recognition [2]. It is an image processing tool rather than an OCR framework.
- Amazon Rekognition: It is another computer vision tool developed by Amazon. The framework uses deep learning technology to identify objects, images and faces [2].
- OCR Space: OCR Space is a software development kit used for getting important information in images but not at the level of Rekognition or Vision API [2].
- Pyocr: This is a python open source framework with OCR tool wrapper. Pyocr is often used as a wrapper for Google's Tessaract-OCR and Cuneiform. It can read all image types supported by Pillow. It also supports bounding box data [2].
- Tessaract-OCR: This is a Optical Character Recognition engine by Google for various operating systems. It was developed at Hewlett Packard Laboratories. It is considered one of the most accurate open source frameworks at the moment.

1.2. Hypothesis

PDFs with computer generated texts can be converted to images with the help of existing python frameworks. Bounding Box technique can be used to extract characters from images and a Convoluted Neural network can be used to learn ascii alphanumeric characters. The trained model can be used to predict the extracted characters in images.

1.3. Why we should solve the problem

The goal of our project is to improve the existing technologies and frameworks that utilize Optical Character Recognition to predict printed text in images. The current APIs and frameworks are image processing tools that extract important information from images. They required preprocessed data. We have developed a pipeline from

scratch that converts computer generated text in PDFs to text files.

2. Approach

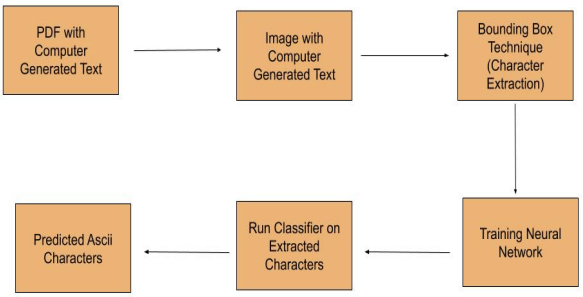


Figure 1: Describes the Pipeline used

2.1. Dataset

The specific dataset we used to train our character classifier was found online at the following link: http://www.ee.surrey.ac.uk/CVSSP/demos/chars74k/. From this site, we specifically used the EnglishFNT.tgz dataset. This dataset consists of 62 classes of machine printed alphanumeric characters (0-9, A-Z, a-z). Each class had over 1,000 sample images consisting of different types of fonts, including bold and italic styles. Each of these images was initially 128x128 pixels.

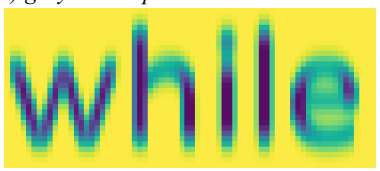
This dataset provided us with a diverse set of sample characters, allowing us to train a robust classifier that could handle differences in font and lettering styles across a variety of potential inputs.

For testing our pipeline as a whole, which includes character bounding box calculation and classification, we used a sample PDF from a school essay. This same page of essay was converted to various font types and tested on our pipeline to see how accurately we could detect and classify the text.

2.2. Preprocessing

Our pipeline begins with preprocessing steps on the input pdf. The first step in the process is opening the pdf as an image, and processing that image. To begin, a 9x9 gaussian kernel was applied to the input text. This gaussian blur had variable sigma in the x and y directions: sigma_x=0.1 and sigma_y=1.83. The reason for this was to be able to detect the full range of letters vertically, while maintaining a clear boundary between the letters for segmentation horizontally. For example, the character "i", is not detected well as a complete contour due to the space between the dot and the body of the "i". By blurring this character vertically, we connect these two portions and detect it as a single contour. At the same time, the small sigma in the horizontal direction maintains a clear divide between characters, keeping them detected as individual blobs. An example of this blurring and its effects are shown below.

a) grayscale input no blur



b) after blur image

Figure 2: Shows the effect of blurring the input image, allows for full detection of character blobs.

Following the blurring of the image, thresholding was necessary in order to create individual blobs in the image for contour detection, as seen in figure 2. The thresholding was not adaptive and was based on pre-set threshold values. This is one area for improvement in the future, to handle more complex input and build a more robust pipeline an adaptive thresholding method should be implemented.

Figure 2: Shows the effect that thresholding had on the input image.

Following this thresholding step, the input pdf image was now ready to enter our OCR pipeline. The subsequent steps of the pipeline will be discussed in the following sections.

2.3. Frameworks and Tools

Our implementations make use of the following tools and frameworks:

- OpenCV
- Pytorch
- Numpy
- Skimage and Sklearn
- Matplotlib

All blurring, thresholding, contour detection, and resizing processes were implemented using OpenCV.

2.4. Architecture and Pipeline

2.4.1. Character Detection Pipeline

Following the thresholding step in the preprocessing portion, our pipeline commences by finding character contours. To do this, we used the OpenCV function 'findContours'. This function outputs bounding boxes around blobs in the binarized input image. Using special arguments, only contours with no parents were returned by the function. An example of the output can be seen below in figure 3.

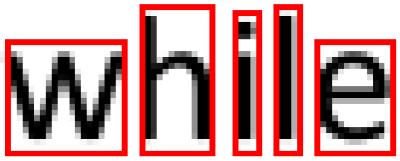


Figure 3: An example of the bounding box output from findContours drawn around the characters of a word.

Following the identification of contours and their locations, our pipeline extracted these regions of interest in order to classify them. Due to the variable nature of the bounding box shapes, which depends on a variety of factors including font size, letter, and capitalization, we needed a way to standardize the shapes for classification. To do this, we implemented a function that would crop the characters from the original image, pad them to get a similar aspect ratio as our training data, and finally resizing to 128x128, the same size as our training images. In addition to these steps, there was some image blurring and thresholding that eventually led to the output that is demonstrated in figure 4 below.

Figure 4: Two examples of the output of our character detection pipeline. These 128x128 images are made to look similar to our classifier training data. They are the inputs for character classification in the next part of our OCR pipeline.

The architecture for the classifier that these images are fed into is described in the sections below.

2.4.2. CNN Architecture

Due to GPU RAM constraints, the images had to be resized to 32x32 pixels from the original 128x128 size. Additional image preprocessing measures taken were batch standardization with mean 0 and unit standard deviation and image thresholding to ensure a binary pixel representation.

(a) *(b)* *(c)*

Figure 5: Some of the preprocessing methods done for input images into the model: (a) 128x128 original image, (b) resized to 32x32, (c) thresholded 32x32 image

The CNN architecture was based off of the initial architecture used in homework 5. The initial model had 2 convolutional layers and 2 fully connected layers with a single dropout layer after the first fully connected layer and max pooling layers after each convolutional layer. Architectural experimentation included adding another convolutional and fully connected layer, adding a batch normalization layer, and adding more dropout layers. Model hyperparameters that were experimented with were a stepped learning rate schedule, an increased number of epochs with GPU computing power, and an increased batch size.

Model	Architecture
Base	2 Conv Layers, 10 Epochs
Base+	Base w/ 20 epochs, Batch Norm, Dropout after Conv1
3L-Base	3 Convolutional Layers w/ Batch norm and dropout
3L-SLR-FDO	3L-Base w/ dropout after every layer and Stepped Learning Rate every 5 epochs for 20
3L-SLR15-Norm	3L-SLR-FDO w/ thresholded and normalized images, learning rate decrease every 15ep
3L-SLR15	3L-SLR-FDO w/ thresholded images, learning rate decrease every 15ep
CPU-128x128	Non resized images ran on CPU

Table 6: Code names for the models in the table below

Model	Acc
Base	.8289
Base+	.8410
3L-Base	.8557
3L-SLR-FDO	.8422
3L-SLR15-Norm	.8694
3L-SLR15	.8717
CPU-128x128	.8774

Table 7: Models tested and their validation set accuracies

The final model chosen was 3L-SLR15-Norm because even though the non-normalized version scored slightly better on the validation data, it is important to normalize the image and the difference in accuracy is miniscule. It was not practical to choose the CPU model as it took about 5 hours to train. The 15 represents the step size of the learning rate decrease. In figure 10 you can see where the first learning rate change is by the sharp decrease around epoch 15.

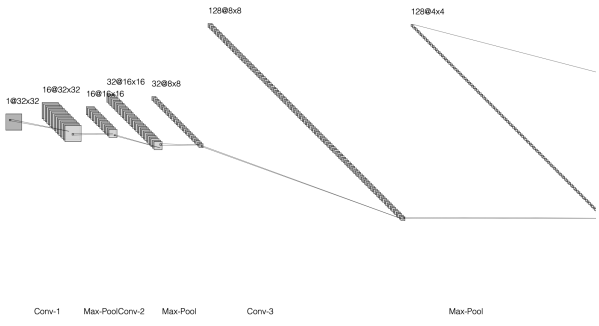


Figure 8: Convolutional layers diagram.

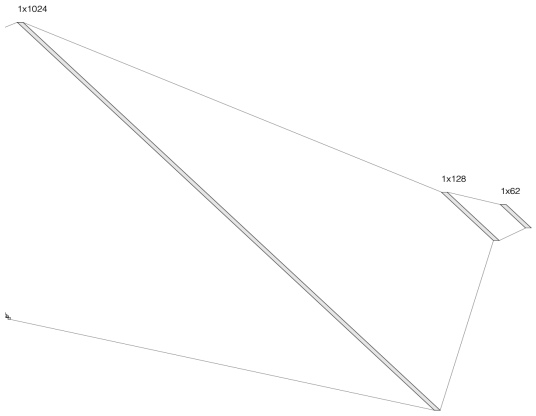


Figure 9: Fully connected layers of model

These diagrams are showing the network for one image, the actual model trains on batches of images. As seen above in figures 8 and 9, the final model had 3 convolutional layers which introduced more filters each layer and halved the size of the images by using max-pooling layers following each convolutional layer. The activation function used was the ReLU. There are 2 hidden fully connected layers which decrease in size until the output layer of length 62 giving a score for each class in the dataset.

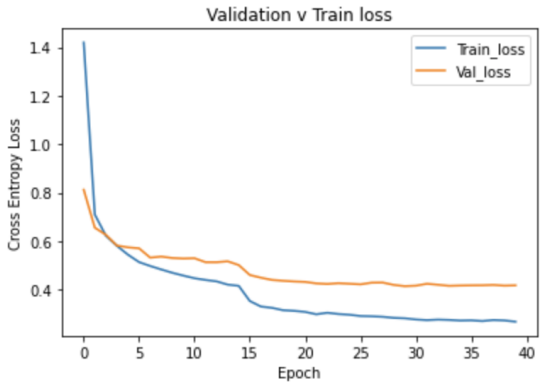


Figure 10: Cross entropy loss during training for the validation and train data

On the holdout test data, the model achieved an accuracy of 85.33%.

2.5. Mathematics

As discussed in our CNN architecture, we have 3 convolution layers, each followed by batch normalisation, dropout and relus, and three fully connected linear layers following these convolution layers.

Each convolution has a kernel size of 3, stride of 1 and padding of 1, these each convolution layer, following the formula:

$$\text{output width} = \frac{W - F_w + 2P}{S_w} + 1$$

$$\text{output height} = \frac{H - F_h + 2P}{S_h} + 1$$

Figure 11: Formula for output size of image (Filter Width: Fw, Filter Height: Fh, Padding: P, Image Width: W, Image Height: H, Stride: S

has input, output channel parameters as 1 and 16, 16 and 32, 32 and 128 respectively for each convolution layer, causing an input of 128*4*4 number of neutrons for the first fully connected linear layer, 256 for the next, and, finally, 62 classes in the final layer's output.
We use dropout with a probability of 0.3 and batch norm with dimension as 16.
Finally, our model is trained with a learning rate of 1e-3 and 40 epochs and has a testing accuracy of about 86%.

2.6. Details of Implementation

2.6.1. Character Detection Pipeline Details

There are a few implementation details in the character detection side that were not discussed in the passages above. The first aspect is the sorting of bounding boxes in the proper order. This was done by using the output from findContours which included the x and y positions of the top left corner of the box and its height and width. Once the bounding boxes were sorted by line, and within each line, this order was kept and passed to the classifier.

In addition to the detection of characters, we needed a way to find the location of spaces between words. To do that we again utilized the location data of the bounding boxes. Specifically, the distance between adjacent bounding boxes in each line was calculated. the distance between adjacent characters should be much smaller than the distance between the last character of one word and the first character of the word that follows after a space. An example of the space distribution between characters for a given line can be seen in the histogram below in figure 12.

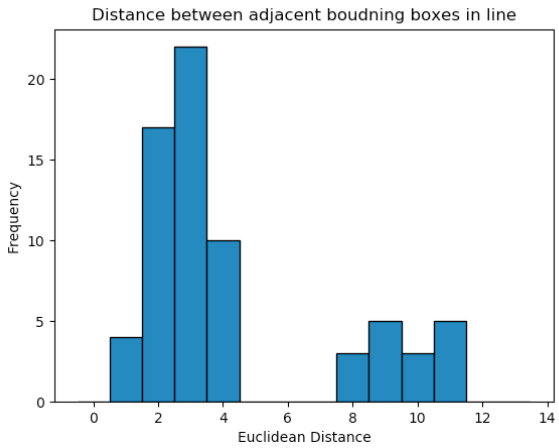


Figure 12: Shows the distribution of distances between adjacent bounding boxes in a line of text from a pdf.

As can be seen in the figure above, two distinct distributions are shown. On the left, are distances between characters within one word, and on the right are distances between the ends of each word. Using this data for each line we were able to calculate the threshold distance that dictates whether there is a space between two characters or not. The indices of these locations, where the distance is greater than this threshold, are the locations of spaces in the text. This information was passed to the classifier, along with the characters for classification, and the final output included these spaces at the precise locations.

3. Result (Angad)

Our preprocessing output helped us extract all alphanumeric characters and was able to maintain and keep track of the spaces. Each of these extracted characters are passed into our trained CNN model in order to classify each of them either as a number or an upper or lowercase letter. It seemed our trained model was able to classify most of these character boxes correctly, however, there were a fair amount of miss hits for the classifier, where many similar characters, such as the letter 'o' and the number '0', and some of the original text, including punctuations and special characters, were lost in our final text output. As we began testing on less homogeneous

text, such as different fonts and styles, there was an increased amount of such misses.

3.1. Preprocessing Output

Our preprocessing pipeline converts PDF to images and uses the bounding box technique to extract computer generated characters from images. At the end of our preprocessing stage, we get a 5D array as input for our classifier. The 5-dimensions include: The number of pages in the PDF, the number of lines per page, the number words per line and 128x128 image matrices for each character in a line.

3.2. Input and Output (Model)

Sample Input:
https://drive.google.com/file/d/1_aJUGePmV_RxOKD1KmXfbz4JuzM4mezr/view?usp=sharing

Sample Output:
https://drive.google.com/file/d/1-4SL3aB4D5rcMs7sEHt07BG4tsHnKDjx/view?usp=sharing

3.3. Analysis

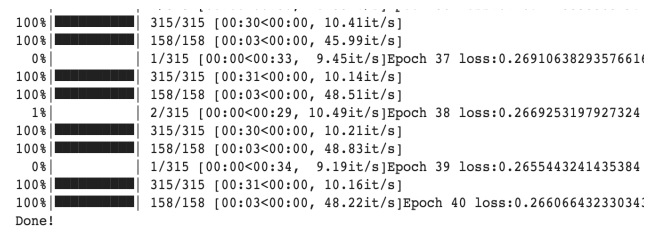


Figure 13: Training Loss after 40 Epochs

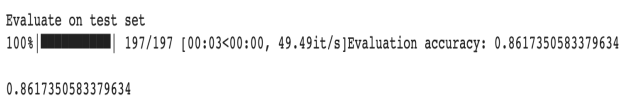


Figure 14: Testing Accuracy

Our bounding box implementation extracts each character in a fairly homogeneous manner (128x128) with negligible errors. Our dataset of 62,000 images was broken into training, validation and testing sets. Our model was fairly robust and achieved an accuracy of 86%. After training our dataset with 40 epochs, we were able to minimize the loss to 0.26.

Our Model detects and classifies most characters perfectly, and seems to have problems with characterising between very similar characters such as the letter ‘o’ and the number 0, or the letter ‘e’ and the number 3.

In addition to this, our model fails to classify punctuation at the moment. Although we can detect punctuation marks using our character detection pipeline, currently our classifier is not trained to classify those characters. This is something that could be solved quite easily by just expanding our training dataset to include punctuation marks such as commas, periods, quote marks etc. This is one future direction that we plan on pursuing with more time.

4. Conclusion

Our program does meet our hypothesis of being able to extract characters from a pdf, classify those characters fairly accurately with a Convolutional Neural Network and output the text into a text file with the proper spacing. Our initial Architecture of 3 convolution layers, max pooling layer, dropout layer, and ReLU activation function met our requirement and gave us a test accuracy of 80%. After tuning our hyperparameters we were eventually able to reach our desired accuracy. We were able to successfully utilize the tools available to us (python frameworks, Google Colab and open source code for bounding box technique) to prove our original hypothesis.

With more time and tools, we could improve the accuracy of our model by developing a deeper neural net, with more layers and train on less homogeneous data. With that said, future directions for our pipeline could include handwriting detection and classification from images. This would require adjustments to our training dataset as well as a more robust and advanced character detection system.

Video Presentation :
https://drive.google.com/open?id=1xoVhyrYRB2JQ1U-7FsEXV8wRgioxa1q1

References:

1. Optical Character Recognition https://searchcontentmanagement.techtarget.com/definition/OCR-optical-character-recognition
2. Building your own OCR https://medium.com/@balaajip/optical-character-recognition-99aba2dad314
3. OpenCV OCR and text recognition with Tesseract https://www.pyimagesearch.com/2018/09/17/opencv-ocr-and-text-recognition-with-tesseract/