Group 146 Project Report: textOCR

Dhruv Chand, Roshaan Quyum, Arin Khandelwal
{chandd9,quyumr,khanda3}@mcmaster.ca

1 Introduction

We are creating an application that extracts text from an image, otherwise known as Optical Character Recognition (OCR). It is a multiclass classification task using images with text on it as its input data. It can also be seen as a single label classification task where each data point will be classified into only one class representing the character that it is. We will create a command line application that takes an image with English text as input and outputs the text on the image into the terminal. It involves implementing a number of different parts of the machine learning and model training process such as image preprocessing, text recognition, and character recognition. We shall use machine learning libraries such as OpenCV, NumPy, Scikit-Learn and PyTorch to implement them.

2 Dataset

- from the sheet. check the data set from. refer to the new data set.

We chiefly used two datasets: OCR-Dataset and Words MNIST.

OCR-Dataset consists of the 26 letters of the English alphabet in both lower and uppercase along with all 10 digits; across the 62 classes, there are roughly 210 000 images in total. The data uses 3475 different fonts from publicly available Google Fonts. It is the dataset we trained our model on.

There are a number of preprocessing steps done to the images from this dataset. Images are converted to a specific type (`np.uint8`) to ensure consistency in the following preprocessing steps. Then, all white pixels on the edge of the letter are cropped out (the cropping is similar to a bounding box being placed around the letter/digit and then all pixels outside of it removed from the image). The image is then scaled to have dimensions of 64x64, and finally, it is turned into a tensor. Before being fed to the model, a corresponding list of labels is generated from the provided labels for more convenient access. After this preprocessing, the images from this dataset can be inputted to the model. This preprocessing here assumes the image contains a single character; however, for images with more than one character, extra preprocessing was needed.

The Words MNIST dataset contains about 10 000 images of various words. All characters from the alphabet (lower and uppercase), all 10 digits, as well as an assortment of special characters/punctuation are in the words in the images. The images have variable sizes and need to be resized to be uniform. The images mostly come from scanned documents and synthetic generation Data was synthetically generated to have lesser seen characters included in the dataset. Some of the images were labelled manually and some of it was labelled using tesseract OCR and then manually checked after for errors. Our OCR model is built only to handle individual characters; so before running the preprocessing from the previous paragraph on it, the characters in each word needed to be segmented. We had two different attempts at this and will describe both.

Our first, less successful segmentation, first grey scaled the image, applied median blur to remove "salt and pepper" noise, applied inverse binary threshold using Otsu's method, and dilated the image to make the characters larger in the vertical direction. We then found the contours of each character and drew bounding boxes around them. The dilation was meant to preserve the gap between characters, which manifested horizontally, and remove any "intra-character" space, such as the hole in an "a" or the large amount of space in the column a "u" takes up so that all characters looked roughly like rectangular blobs. Finally, we used the dimensions of each bounding box to crop out the characters from the original image and return an array of images containing each character. One

large flaw of this approach is that if the dilation did not remove all the space in a character's rough column location, then multiple contours would be drawn in that column. This would mean, for example, that both the bounding box containing "a" as well as the circle in "a" would be returned as distinct characters. With the variation in font, this occurred quite often with "o"s and "a"s.

The second, more successful segmentation function implements the Potential Segmentation Columns (PCS) method from the paper: A New Character Segmentation Approach for Off-Line Cursive Handwritten Words. The same rough process is done as before with the exception of blurring before thresholding and instead of dilating the image, we use Zhang Suen's method to thin the characters. After thinning, all columns of pixels in the images are scanned. If there is less than or exactly 1 pixel (this was generally the PCS threshold/limit for the images in the Words MNIST dataset) of white (meaning less than 1 pixel of character), then the column is marked as being a PCS. After the image is fully scanned, the array of marked segmentation columns is looped through. For each set of consecutive PCSs, the middle column of the set is chosen as the segmentation column and the rest are unmarked. Finally, the original image is cropped according to these columns and the individual images are returned. This method worked much better than the first one and is near guaranteed to work when there are gaps between characters. When there aren't any gaps, it still performs well and better than the first method, but clearly not as good. This method was chosen for character segmentation due to its consistency (it is easier to find out why it fails) and better quality.

3 Features and Inputs

- diff features and inputs for preprocessing.

The inputs to the current model are 64x64 dimension images. In our final version of our model, no feature engineering, representation learning, feature selection, or augmentation was done. The preprocessing as described above was done and then the image(s) were fed into the model. We chose to do this as this is how images are normally fed into a convolutional neural network (CNN).

Before we finalized our model, we used a different set of features. Originally, we generated Histogram of Oriented Gradients (HOG) features to feed our model. This is feature engineering technique that is used for object detection. Due to this, we chose to use it for our early models. The HOG features were varied by changing the parameters that define how they are generated. For example, the number of cells/boxes that features are calculated for (gradients are calculated for each cell and cells represent boxes larger than pixels that the image can be split into) can be changed and this can drastically change the size of each individual feature. For example, the cell size being decreased can greatly increase the number of features since much more gradients are being calculated for the image (a smaller cell size means more cells are needed to cover the image and gradients are calculated for each cell). Conversely, the cell size being increased can greatly decrease the size of each feature since less gradients are calculated.

4 Implementation

Initial Models

Our inital model Implementation was using PyTorch and is based on a feedforward neural network architecture. It was trained using the HOG features with mini-batch gradient descent for optimization and cross-entropy loss for learning. We landed on using a stochastic gradient descent (SGD) optimizer with a cross-entropy loss function initially as it allows for quick training of our model. We tested our model using an Adam optimizer instead, but it did not improve results much, while slowing down the training significantly. This model consisted of 4 linear layers containing 896 total hidden neurons (512 on layer one, 256 for layer two, and 128 for layer three) interspersed with ReLU activation functions to prevent negative values. The network also contains dropout layers to prevent overfitting.

Training was conducted on a 80/20 training/testing split, with a batch size of 64 over 500 epochs, with a early stopping patience implements of 100 epochs which halts training if validation accuracy stop improving. Through the training, our model logs the training loss, validation loss, and validation accuracy every 10 epochs to allow us to monitor model efficiency. Our current model provides a 68% validation accuracy. This is a not perfect score, but performs decently and can accurately classifies characters most of the time.

We initially used the random baseline to compare our model too. With 62 classes, random guesses would be $1/62 = 1.6\%$. This model performs much better than random guessing, but can clearly

still be improved on. We will also use this model's results as another baseline for our final model.

Final Model

Our final model implementation was using PyTorch based on a multi-layer CNN architecture. It operates on grayscale image input and uses mini-batch gradient descent for optimization with a cross-entropy loss function. We decided on this architecture because convolutional layers are very effective at extracting spatial features and image recognition tasks, fitting into the project requirements nicely. The network consists of three convolutional blocks with 32, 64, and 128 filters. Each block contains a convolution layer, a batch normalization layer to help stablize training, a ReLU activation function to ensure non-negativity, a max-pooling layer for spacial downsampling, and dropout layers to reduce overfitting. The output of these layers are fed into two fully connected linear layers containing 256 hidden neurons and a final output layer corresponding to the number of classes within our dataset.

Training was done with an 80/10/10 training-validation-test split where the dataset was randomized before being split. A batch size of 16 was used with a 0.03 learning rate and weight decay of 1e-3 for the SGD optimizer. Similar to the previous model, early stopping is implemented to stop learning when validation accuracy stops improving. Using this model, we achieve a test accuracy of 90%. It clearly performs much better than the previous model.

Once again, we compare this model to some baselines. It is clearly better than our previous model and the random baseline. We will also compare it with other state of the art OCR models. The models we will compare against, unlike our model, are much more complex and can take whole PDF documents as input. We use a table sourced from a report from Mistral AI that compares their in-house OCR model with others. They compare their model against others on a variety of benchmarks including how well tables and math can be scanned, but we will use their benchmarks for recognizing words from different languages only.

Language	Azure OCR	Google Doc AI	Gemini-2.0-Flash-001	Mistral OCR 2503
ru	97.35	95.56	96.58	99.09
fr	97.50	96.36	97.06	99.20
hi	96.45	95.65	94.99	97.55
zh	91.40	90.89	91.85	97.11
pt	97.96	96.24	97.25	99.42
de	98.39	97.09	97.19	99.51
es	98.54	97.52	97.75	99.54
tr	95.91	93.85	94.66	97.00
uk	97.81	96.24	96.70	99.29
it	98.31	97.69	97.68	99.42
ro	96.45	95.14	95.88	98.79

In this table, the `fr` (French), `de` (German), and `es` (spanish) rows are what we will focus on since they use the same script, latin, as our data. As can be seen, the lowest accuracy in this group of three rows is 96.36% from Google Doc AI. Every other models performs better than this.

There is one main reason and a smaller second reason that we believe to be the cause of our model performing worse than these models. The first is the size of our training data. Although these models are closed source and so we cannot know exactly what they have been trained on, given that they can extract text, tables, math, etc. from PDF documents, we can likely safely assume that they have been trained on much more data that has much more variety than English characters than our model has. From the outset, our model was not planned to be so advanced and we made our architectural and data decisions with that in mind. The models in Mistral AI's report are meant to be used by large corporations to reliably automate document processing. They developed their own models as such. The second, smaller reason is computation power. We had access to McMaster's two H100 GPUs to train our model, but with more GPUs, we could have used many more layers as well as some of the techniques learned in class such as residual blocks to boost our accuracy.

5 Evaluation and Progress

The Main evaluation metrics that we tracked to measure how well our models were doing were Character Classification Accuracy and Word Recognition Accuracy.

We
evaluation metrics that we used are
Character Accuracy per character
and
Word recognition accuracy

Some stuff planned aspects of our application from the progress report did get changed as we did not develop the "command line application" plan for the front end that we had originally envisioned. This is due to us not having enough time and therefore choosing to prioritize improving our model structure. Instead, we create a python file that can be run within your terminal with a provided image path, which returns the predicted text in the image.

From the project report we followed most of what we set out to do. We obtained the target accuracy that we set out for of 90%. We expanded our model to include more non linear layers, such as convolutional and pooling layers. We visualized our metrics more to be able to understand and compare the models we created better. We slightly diverged from the plan of using a pretrained models or a existing library, instead we implemented a known method for segmenting text obtianed from online research into own code.

6 Error Analysis

- Need to figure out

Team Contributions

7 Figures and Tables