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 created a python 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 used machine learning libraries such as OpenCV, NumPy, Scikit-Learn and PyTorch to implement them.

2 Dataset

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. It is licensed under CC0: Public Domain. The data is kept within folders with the folder names being the class names. For example, images with the uppercase "A" in them are stored with in the "A_U" folder and images with the lowercase "a" in them are stored in the "a_L" folder. The names of the images themselves are the same as the folder name with an underscore, a number, and the image type appended to them. Using the first part of the example from above, an image stored within the "A_U" folder may have name "A_U_x.png" where x is some number. One of our images is named "A_U_1003.png".

There are a number of preprocessing steps done to the images from this dataset. Images are first 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 being removed from the image). The image is then scaled to have dimensions of 64x64, and finally, it is turned into a tensor that is accepted by PyTorch's `torch.nn.Conv2d` function. These tensors are saved to disk so that the preprocessing does not have to be repeated each time the model is trained. Before being fed to the model, a corresponding list of labels is generated from the provided labels and saved to disk for more convenient access similar to the features. These labels are converted from strings to numbers using scikit-learn's `LabelEncoder` as well. After this preprocessing, the data can be inputted to the model. The preprocessing as described so far assumes every single image contains a single character. So, this preprocessing works for the images in this dataset; but, we are aiming to use this model trained only on such images on images with more than one character as well. Clearly, extra preprocessing is 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 labels are given in a dictionary in a `json` file. The images are not contained within any folders nor have any structure; they are all located in one directory. The names of each image are some number with a dot and the file image file format appended to the number. For example, "0.jpeg" contained the word "BE" (case sensitivity is intended here). 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. The images have variable sizes and need to be resized to be uniform. Be that as it may, our OCR model is built only to handle individual characters; so, before running the preprocessing from the previous paragraph that makes images suitable for being passed to the model, the characters in each word need to be segmented. We had two different attempts at this and will describe both.

Our first, less successful segmentation, grey scaled the image, applied median blur to remove "salt and pepper" noise, applied inverse binary threshold using Otsu's method to automatically find the optimal threshold value/limit, 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. We do this so that the contours algorithm finds a single contour (the contour of the rectangle blob) for each letter, allowing us to detect each individual character. Finally, we used the dimensions of each contour to draw a bounding boxes them and crop out the characters from the original image using them. Hence, we 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 (PSC) 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 PSC 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 PSC. After the image is fully scanned, the array of marked segmentation columns is looped through. For each set of consecutive PSCs, 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

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 91% on characters. It clearly performs much better than the previous model. We also tested it on the Words MNIST dataset using the PSC algorithm for breaking the images of words into an array of images with characters. The accuracy of this when tested against the whole dataset was 53.72%. When loosening the accuracy measure to treat words with one character being classified wrong as correct, the accuracy is 78.30%.

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

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

Character Classification Accuracy

We evaluate the character classifcation accruacy by calculating the accuracy, precision, recall, and F1-Score after training our models, and comparing it to eachother and a basic "majority class" baseline. (predicting the label for a image based on their appearance in our dataset).

Examining our HOG model, we can see the accuracy, precision, recall, F1-Score, and support values obtained after training in (link). Looking at (link) we can see that this model achieves an overall accuracy of 67% on the test set of 682 samples. The macro-averaged metrics show precision of 68% and the weighted averaged is 70%. This represents a strong improvement over the baseline and demonstrates that the model has learned meaningful patterns from the HOG features. This beats the baseline on our dataset which would have a 1.6% accuracy, as the data the HOG model is tested on contains 62 classes, each with 52 images each, giving each class a 1/62=1.6% chance of being chosen.

Examining our CNN model, the accuracy, precision, recall, F1-Score, and support values obtained after training can be viewed in (link). Looking at (link) we can see that this model achieves a overall accuracy of 90% on the test set of 21023 samples. The macro-averaged metrics show precision of 91% and the weighted averaged is 91%. This easily beats the baseline on this dataset which is ?? accuracy, as the dataset contians 62 classes, each with approximately 3475 images each, giving each class a approximate probablty of 1/62=1.6% chance of being chosen.

Word Recognition Accuracy

Using the MNIST dataset containing 10K printed text words, we chose 6001 images from it to test our model on. These images were chosen from which ones resulted in a "valid word segmentation". We determined this by running our word segmentation code upon it, and if the number of segmented images returns from it matched the amount of letters in the images label, it was a valid word segmentation. This means that our word segmentation algorithm failed to accurately segment 3999 of the images.

Running our CNN model on these 6001 images of printed text resulted in a 3224 words predicted correctly and therefore achieved a 53.72% accuracy. Another metric that we tracked was how many words did the model calculate "partially correct", in other words, if we allow 1 character to be incorrectly predicted. The results obtained are that the model predicts 4699 of the words "partially correct", giving a accuracy of 78.3%. This demonstrates that our model predicted most words correctly, but fails on just one word in the model.

6 Progress

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.

7 Error Analysis

Results and Analysis

Looking the confusion matrix generated for validation set of HOG model, seen in (link). (link) shows both the classes which are most misclassified and correctly classified. Looking the graph we can see that the classes '0'/'O', '1'/'I', and 'n'/'r' are often misclassified as each other, which makes sense as they share very similar shapes, and therefore will have very similar HOG features.

The confusion matrix generated for the CNN model can be seen (link). (link) shows a similar pattern as the HOG model, but is ran with a significantly larger dataset therefore results seem quite different. Looking at the graph, we can continue to see the pattern of classes '0'/'O' being incorrectly classified, once again due to their similar shapes. We can also see new errors such as 'c'/'C', 'l'/'I', or "/".

However, a clear difference found is that the CNN model performs much better at classifying 'n'/'r' and '1'/'I' as there are almost no mix up between the two. This could either be a reason from having a better dataset (typed letters have more clear distinct features), because of the CNN architecture, or both.

Potential Solutions

To address the issue of '0'/'O' and uppercase/lowercase misclassifications, one potential improvement is to increase the resolution of the input images. Our current 64×64 inputs may be too low to capture the subtle stroke differences between similar characters such as O vs 0 or C vs c. Increasing this to 96×96 or higher could give the model more detailed visual information to learn from. Incorporating data augmentation during training would also help the model generalize better by exposing it to variations in rotation, brightness, and slight distortions. Additionally, reducing the amount of "downsampling", or compression in our CNN architecture may help preserve important fine grained features, allowing the network to retain more shape information needed to distinguish similar characters.

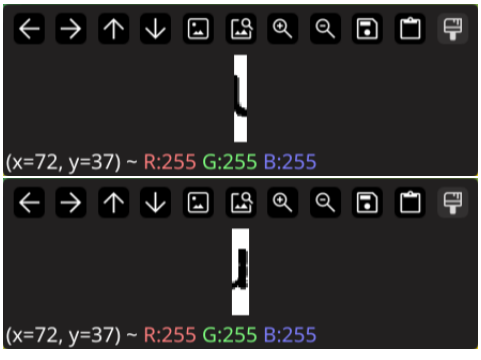
Segmentation Errors

Specifically for the Words MNIST data, one large source of errors was the segmentation used for splitting words up into characters. The dataset contains many images of words with different fonts and thus the gaps between characters also vary. Fonts which have the characters pressed up against each other leads the PSC algorithm treat those two different characters as one character and thus output them in one image. For example, the word displayed here:

is segmented normally until the two "L"s. These two "L"s are segmented into one word as such:

leading to the model misclassifying this image. Another common problem are the cases of "u"s and "n"s. The section of these two letters that connect the vertical segments are often very thin; so, the thinning on characters done in the algorithm often makes that section so thin that the PSC algorithm mistakenly selects columns in between the vertical segments as segmentation columns. We use the example of the word "unsatisfactory"':

Even though "r" and "y" are touching in this image, they are still segmented correctly. However, both "u" and "n" are not segmented correctly.



As can be seen here, both of these characters are segmented into two images, adding 4 extra incorrect characters that are passed to the model. Issues like the two above can and do occur; they result in the model being passed poor data and often incorrectly classifying the words.

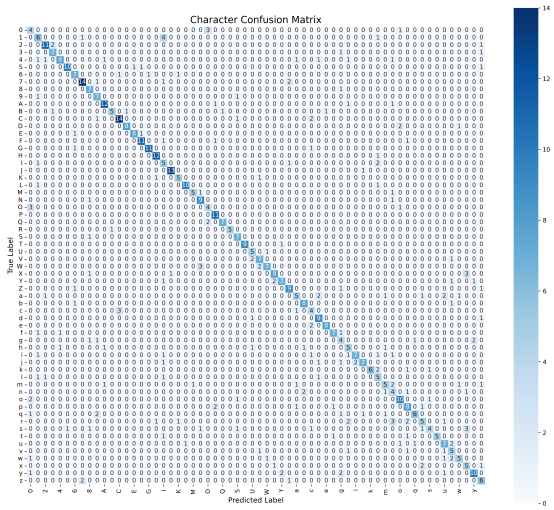


Figure 1: HOG Confusion Matrix

8 Figures and Tables

Team Contributions

Dhruv and Arin developed the current iteration of the Convolutional neural network model, including creating HOG linear neural network, the class for the model, writing the training code, measuring the accuracy of the model, and creating a confusion matrix for it. Also worked on preprocessing for the CNN model, training for both HOG and CNN. Worked on the creation of final script that does the classifying. Lastly, implemented additional preprocessing code to make the data suitable for PyTorch models along with code to ensure that if a GPU is present, it will be used in the training process.

Roshaan contributed to finding the datasets we're using as well as implementing the feature generation and general preprocessing code to input data into the scikit-learn SVMS and PyTorch neural networks. He also attempted to implement multiclass SVMs and in tandem with this, he implemented further preprocessing code to make the data suitable for inputting into the SVM. Lastly, he did troubleshooting for GPU clusters.

Class	Prec.	Rec.	F1	Sup.	Class	Prec.	Rec.	F1	Sup.
0	0.33	0.50	0.40	8	a	0.56	0.38	0.45	13
1	0.46	0.50	0.48	12	b	0.73	0.80	0.76	10
2	0.79	0.79	0.79	14	c	0.36	0.50	0.42	8
3	0.70	0.78	0.74	9	d	0.64	0.82	0.72	11
4	0.80	0.53	0.64	15	e	0.89	0.80	0.84	10
5	0.91	0.77	0.83	13	f	0.78	0.70	0.74	10
6	0.64	0.70	0.67	10	g	0.33	0.44	0.38	9
7	0.82	0.78	0.80	18	h	0.45	0.50	0.48	10
8	0.58	0.88	0.70	8	i	0.78	0.54	0.64	13
9	0.58	0.78	0.67	9	j	0.78	0.58	0.67	12
A	0.80	0.80	0.80	15	k	1.00	0.60	0.75	10
B	0.83	0.56	0.67	9	l	0.38	0.56	0.45	9
C	0.78	0.78	0.78	18	m	0.56	0.42	0.48	12
D	0.89	0.73	0.80	11	n	0.29	0.44	0.35	9
E	0.89	0.80	0.84	10	o	0.67	0.71	0.69	14
F	0.85	0.79	0.81	14	p	0.53	0.62	0.57	13
G	0.92	0.85	0.88	13	q	0.75	0.50	0.60	12
H	0.86	0.92	0.89	13	r	0.50	0.33	0.40	15
I	0.31	0.56	0.40	9	s	0.67	0.31	0.42	13
J	0.93	0.93	0.93	14	t	1.00	0.62	0.77	8
K	0.71	0.71	0.71	7	u	0.54	0.58	0.56	12
L	0.91	0.83	0.87	12	v	0.42	0.56	0.48	9
M	0.83	0.71	0.77	7	w	0.62	0.50	0.56	10
N	0.64	0.82	0.72	11	x	0.42	0.56	0.48	9
O	0.44	0.50	0.47	8	y	0.77	0.62	0.69	16
P	0.73	1.00	0.85	11	z	0.46	0.67	0.55	9
Q	0.88	0.78	0.82	9					
R	1.00	0.83	0.91	6					
S	0.64	0.88	0.74	8					
T	1.00	0.90	0.95	10					
U	0.56	0.83	0.67	6					
V	0.64	0.70	0.67	10					
W	0.78	0.58	0.67	12					
X	0.73	0.62	0.67	13					
Y	0.70	0.58	0.64	12					
Z	0.69	0.75	0.72	12					
Accuracy: 67% **Macro Avg:** Prec=0.68 **Weighted Avg:** Prec=0.70									

Table 1: HOG classification metrics

Class	Prec.	Rec.	F1	Sup.	Class	Prec.	Rec.	F1	Sup.
0	0.69	0.43	0.53	699	a	0.93	0.94	0.94	643
1	0.95	0.87	0.91	688	b	0.97	0.92	0.94	677
2	0.99	0.95	0.97	729	c	0.86	0.92	0.89	669
3	0.98	0.95	0.96	745	d	0.98	0.93	0.96	647
4	0.97	0.95	0.96	712	e	0.95	0.95	0.95	623
5	0.98	0.95	0.96	680	f	0.95	0.94	0.95	620
6	0.98	0.94	0.96	693	g	0.95	0.93	0.94	642
7	0.94	0.96	0.95	699	h	0.97	0.92	0.95	635
8	0.75	0.94	0.84	697	i	0.95	0.95	0.95	647
9	0.96	0.94	0.95	702	j	0.87	0.96	0.91	660
A	0.95	0.91	0.93	688	k	0.98	0.93	0.96	603
B	0.94	0.93	0.94	673	l	0.80	0.53	0.64	636
C	0.91	0.80	0.85	667	m	0.95	0.96	0.95	678
D	0.92	0.92	0.92	695	n	0.94	0.96	0.95	645
E	0.86	0.92	0.89	706	o	0.83	0.83	0.83	666
F	0.94	0.92	0.93	680	p	0.96	0.90	0.93	660
G	0.90	0.92	0.91	706	q	0.94	0.93	0.93	689
H	0.77	0.92	0.84	721	r	0.93	0.94	0.94	642
I	0.56	0.88	0.68	680	s	0.92	0.86	0.89	652
J	0.96	0.84	0.90	693	t	0.97	0.94	0.95	658
K	0.92	0.94	0.93	686	u	0.97	0.92	0.94	626
L	0.93	0.89	0.91	713	v	0.90	0.88	0.89	642
M	0.95	0.88	0.92	714	w	0.92	0.89	0.90	647
N	0.93	0.91	0.92	759	x	0.89	0.93	0.91	629
O	0.46	0.87	0.60	672	y	0.94	0.95	0.94	696
P	0.91	0.92	0.92	723	z	0.91	0.86	0.89	648
Q	0.96	0.89	0.93	728					
R	0.95	0.92	0.94	720					
S	0.85	0.87	0.86	680					
T	0.96	0.91	0.93	694					
U	0.92	0.89	0.91	701					
V	0.89	0.86	0.88	701					
W	0.89	0.87	0.88	689					
X	0.95	0.82	0.88	679					
Y	0.95	0.86	0.90	703					
Z	0.82	0.88	0.85	651					
Accuracy: 90% Macro Avg: Prec=0.91 **Weighted Avg:** Prec=0.91									

Table 2: CNN Model Metrics