Project overview

Optical character recognition is the ability of a machine to identify hand-written or machine written text [1]. It has numerous applications like reading postal addresses, bank cheques, application forms [2], digitizing heritage materials as images [3], automatic number plate recognition [4] and so on. It is one of the earliest areas of AI research and is quite a mature technology today [5]. In fact, there are many commercial software applications today which help digitize texts — for e.g., Tesseract, Adobe Acrobat Pro DC, Microsoft OneNote etc. [6].

Broadly speaking, this is a 3-step process [4]:

- **Image pre-processing** – This involves steps applied to the image to prepare it to be fed to a model. It includes alignment fixes, smoothing edges, binarization, normalization etc.

- **Text-recognition** – This is the crux of the actual problem that we are trying to solve. There are different ways of achieving this. The early photo-cell OCR systems leveraged a matrix matching algorithm that involved comparing an image to a glyph on a pixel-to-pixel basis. More recent feature extraction techniques decompose an image into a set of features and these features are compared with stored values using approaches like nearest neighbor classifiers

- **Post-processing** – This is application-specific. The output can be plain text or a file or it can adhere to the same format as the input. The accuracy of OCR systems can be increased if the output is constrained to a list of words that are allowed to occur (e.g., English language)

A type of OCR called Intelligent Word Recognition [6] enables interpretation of hand-written text of different styles and fonts and that is the main focus of this project.

The aspect of a typical text recognition engine that we will be looking at in detail is shown below [7]:

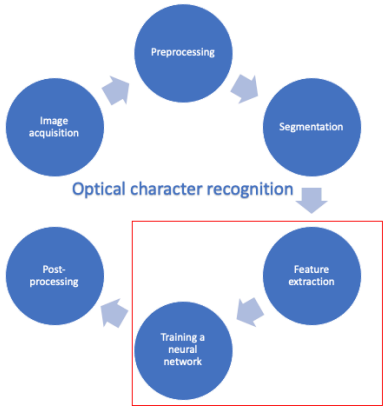


Fig 1. Steps to build an OCR engine (contents in red box will be our focus)

The input image is preprocessed to clean it up and remove noise. This is followed by character segmentation, which puts bounding boxes around text. The next few steps involve interpreting these characters and we will be leveraging a convolutional neural network (CNN) for that.

There are different areas of machine learning that aid OCR. Computer Vision helps with character recognition in images, NLP techniques are used to construct meaningful sentences from words and deep learning algorithms are used for performance improvements [8].

Even though there has been extensive research in this field, there are a number of limitations with current algorithms [6]:

- Document-based: Colored backgrounds/patterns in images, blurry images etc. make it challenging to extract information from images

- Text-based: Alphabets in different languages, the style of writing and fonts used make text recognition challenging

The application of OCR that intrigues me the most is its use in assistive technology for visually challenged individuals and kids with dyslexia [9]. Digital text makes it possible to see and hear words at the same time and that can be tremendously useful to those with reading disabilities.

Problem statement

Given an image (machine generated or hand-written) of a number or an English alphabet, I would like to develop a CNN model that is capable of recognizing it. Once the model has been trained, I would like to test it on the test data set and also on additional custom images. The metrics that I plan to focus on are accuracy, precision, recall and F1-score.

Benchmark model

SpinalNet [29] has been reported to have the highest accuracy of 90.36% [13] on the balanced EMNIST data set. This model mimics the human somatosensory system.

Datasets and Inputs

I plan to use the EMNIST dataset for this project. This is a dataset of handwritten characters converted to a 28x28 pixel image [10]. There are 6 different sets provided in Binary and CSV formats. There are also 6 different types of splits provided for this dataset and I will be using the Balanced split (downloaded from Kaggle) which has train and test data for 47 classes [11]. The balanced split has a more uniform distribution of the classes (~2400 images for each class). Each row in the data represents an image.

Data exploration

Let us print out some information about the shape of the train and test data:

```
Train data shape:  (112799, 785)
Test data shape:  (18799, 785)
```

The first column corresponds to the label and the remaining 784 columns correspond to actual pixel intensities of a 28x28 image. We need to reshape the columns to understand what the data looks like.

The following diagram displays examples of the training data that represent each of the 47 categories. Note that the dataset contains all digits, all alphabets in upper case and only a subset of alphabets in lower case.

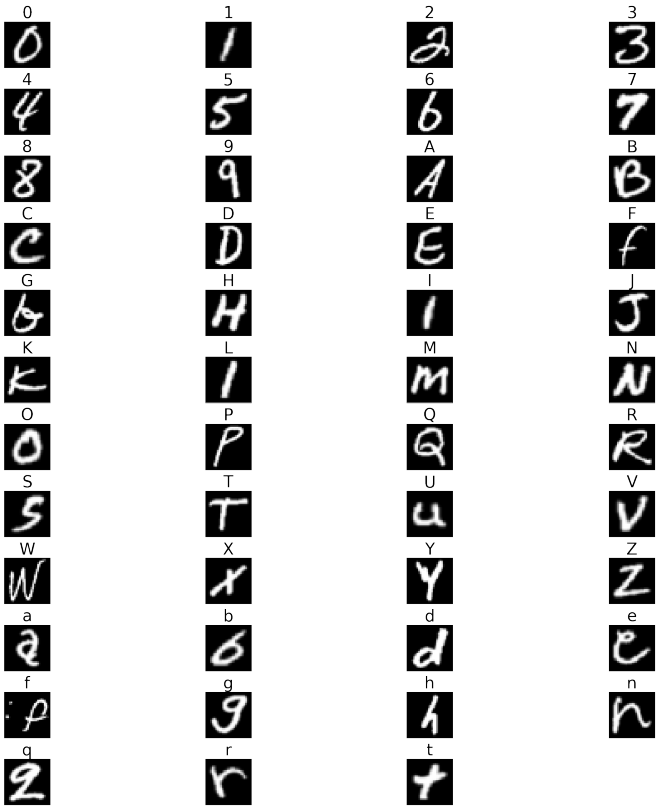


Fig 2. Examples of training data from each category

And we have a uniform distribution of training examples across all labels.

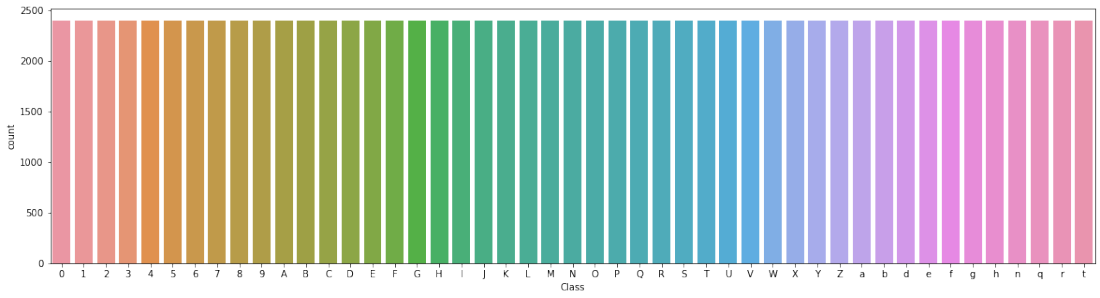


Fig 3. Distribution of training data

We will be normalizing the images to be gray scaled. So, the pixel intensities will be between 0 and 1. The background pixel intensity is 0 and the actual character has pixel intensities value ranging between 0 and 1, as shown below:

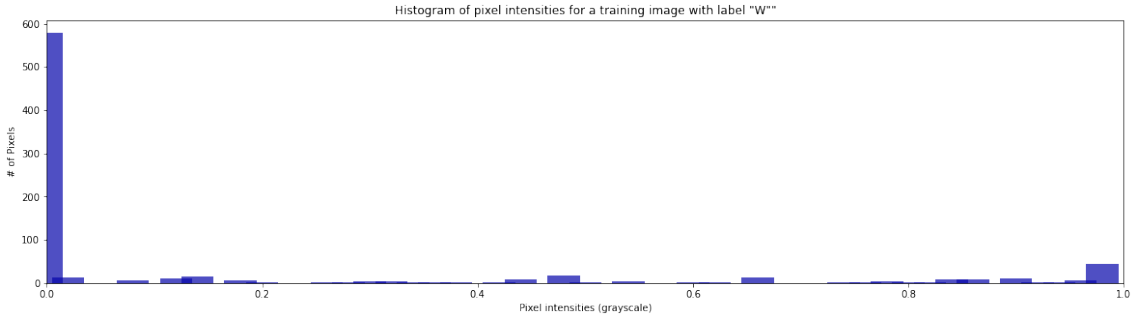


Fig 4. Histogram of pixel intensities for class “W”

It would be good to understand the variability within each class [12]. For this, we find the mean image for all classes (centroid) and look at the distance of our training data from the mean. We use Euclidean distance to calculate the distance between an image and its centroid:

$$\sqrt{\sum_{i=0}^{n}\sum_{j=0}^{m}\left(I(i,j)-Centroid(i,j)\right)^2}$$

Fig 5. Centroid for all classes

To visualize how training images in each category vary from the mean, we can use box plots. They are an excellent way to visualize differences among groups [12].

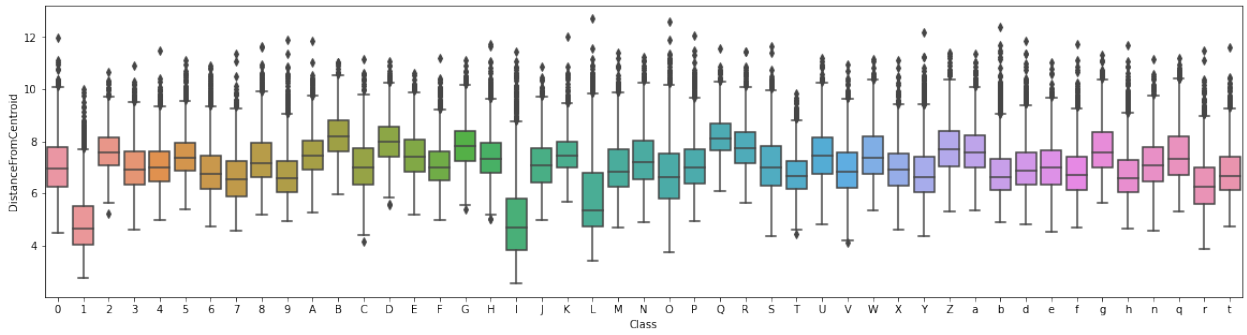


Fig 6. Boxplot of distances of all training images from their respective centroids

The length of the box indicates the interquartile range and tells us how dispersed the data is [13]. The box length is quite high for classes like “L”, “I” and this indicates that there is less consistency in the images representing this class. For example, the following images are examples of those which fall into the box in Fig 6 for “I”.

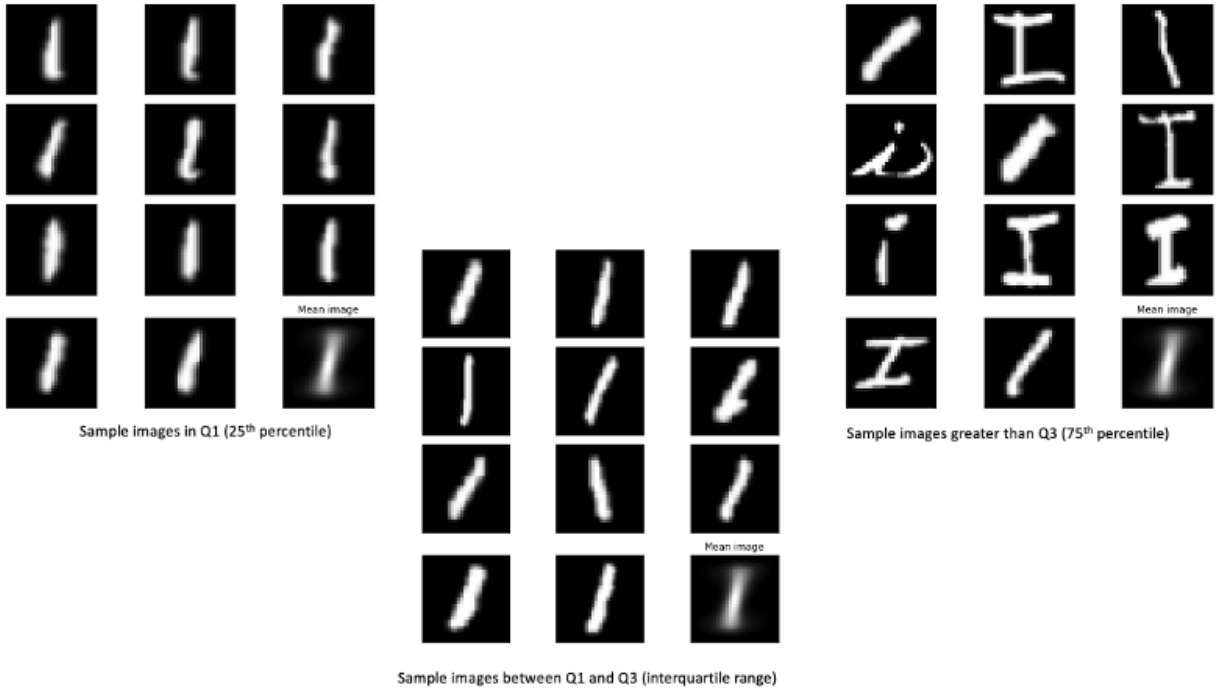


Fig 7. Sample images of class “I” that are at various quartiles represented by the boxplot

In the image at the center in Fig 7, we see that there is a lot of variability, but it is not too far from the centroid (bottom right).

The image on the left in Fig 7 shows examples of “I” that are closer to the mean (Q1) and the image on the right shows examples that are farther away from the mean (> Q3) and this is quite

evident from what they look like. There are indeed many different ways to write the alphabet "I".

Let's look at an example where the length of the box is much smaller — "Q".

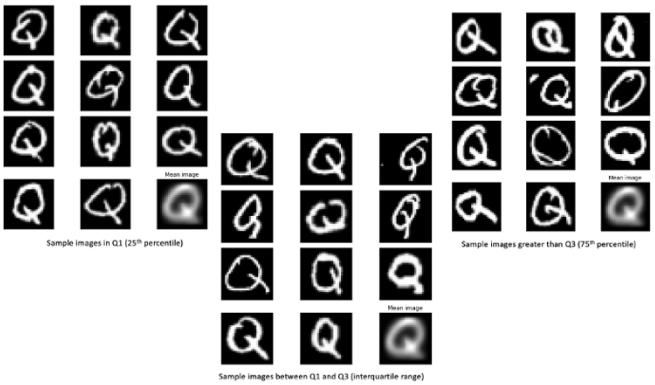


Fig 8. Sample images of class "Q" that are various quartiles represented by the boxplot

There is less variability within the examples of the class "Q". However, most examples are far away from the mean. There aren't too many different ways this alphabet can be represented.
In general, there are many outliers in all classes and we can visualize examples of these outliers to figure out why they are at a great distance from the centroid.

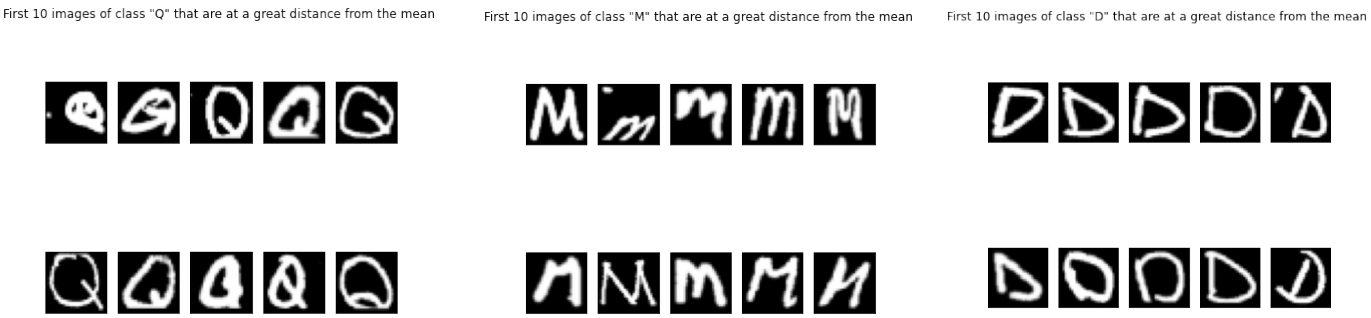


Fig 9. Images that are at greater distances from the centroid for 3 different classes – "Q", "M" and "D"

Another useful visualization is a pair-wise comparison of digits/alphabets [12]. This pictorially shows us how one can be discerned from another. We can plot this by subtracting the centroid of a given class from the centroid of all other classes, as shown below.

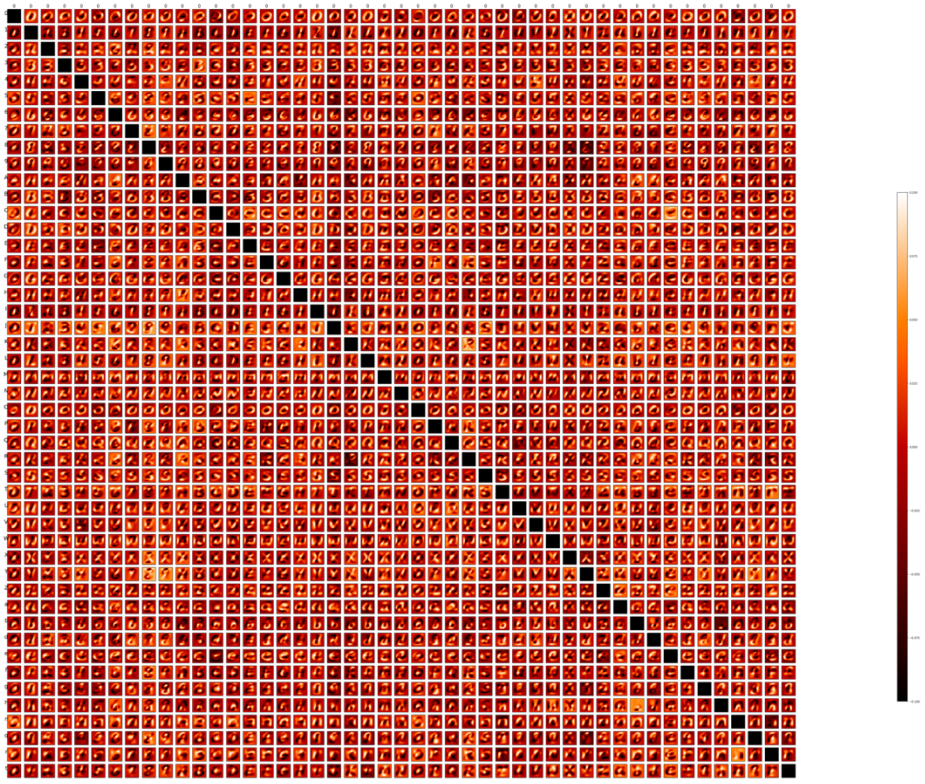


Fig 10. Pair-wise comparison of centroids of all classes

The diagonal is all black because we are subtracting the mean from itself. Regions which are black or yellow make it easy to differentiate one label from the other because they are regions where some features are pronounced. Consider the following example:

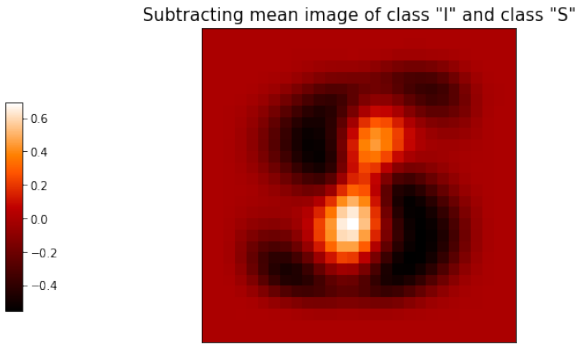


Fig 11. Pair-wise comparison of centroids of classes “S” and “I”

Classes “I” and “S” are very different from each other and this is evident from the difference image. We can identify “S” as the black regions in the image quite easily.

Finally, it would also be useful to understand if the input data can be grouped into clusters that are separable. A visualization technique called TSNE (t-distributed stochastic neighbor embedding) can help us with this [14].

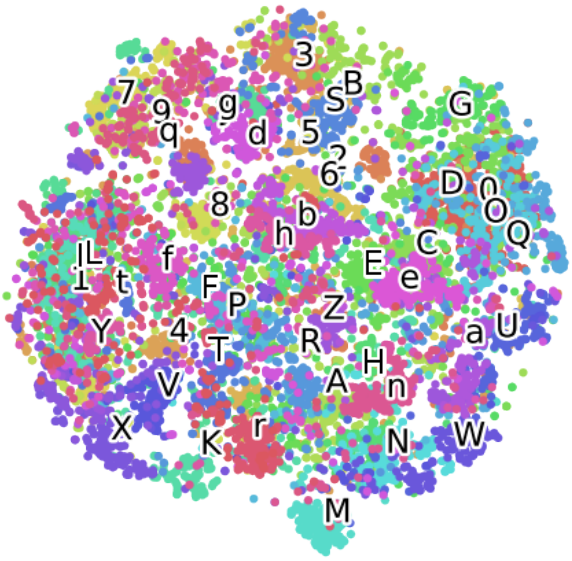


Fig 12. t-SNE on a subset of training data (200 data points from each class)

This plot was generated by picking 200 samples from each of the 47 classes and what this tells is that there is a lot of overlap between the features across classes.

Model selection and evaluation

Convolutional neural networks

When extracted manually, features like corners, edges, blobs, SURF and SIFT are used for image classification [15]. A traditional multi-layer perceptron (MLP) is not a great tool for working with images. This is because, it flattens out the inputs and this results in a loss of spatial information in the image. It is a fully connected network, i.e., each neuron is connected to every other neuron, making it impossible to manage the number of weights that are learnt during training. Having too many weights also results in overfitting [16].

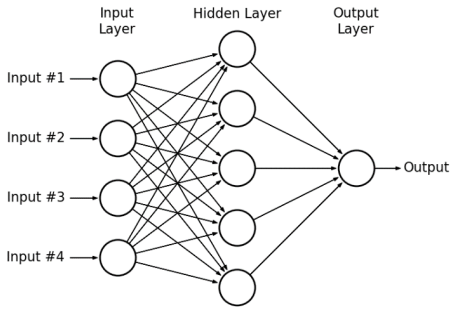


Fig 13. Multi-layer perceptron (source)

We will be using a CNN to automatically learn features from the input images that helps us distinguish one class from the other. CNNs are feed forward neural networks and they are effective in reducing the number of parameters to be learnt by employing filters. These filters help us exploit the spatial locality of images [17]. Nodes that are close together are helpful in identifying a feature and CNNs leverage this spatial correlation of pixels [18]. In fact, CNNs are inspired by the pattern of connectivity in an animal's visual cortex where individual neurons respond to visual stimuli only in a restricted visual field and receptive files of different regions overlap to cover the entire field of vision [19].

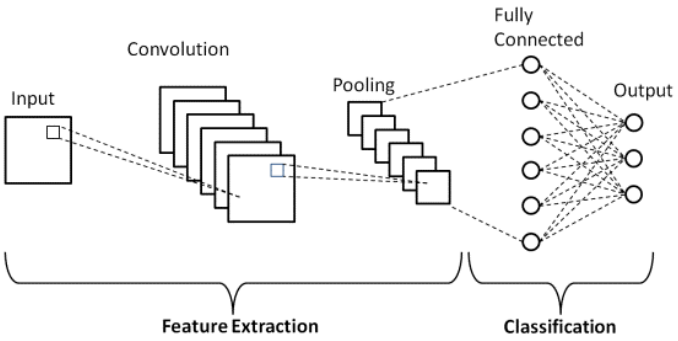


Fig 14. CNN layers (source)

There are 3 different types of layers in a CNN, which are usually stacked [20]:

1) Convolution layer – This is responsible for extracting features from the input images and employs the concept of mathematical convolution. Convolution is a linear operation that involves multiplying a set of weights with the input.

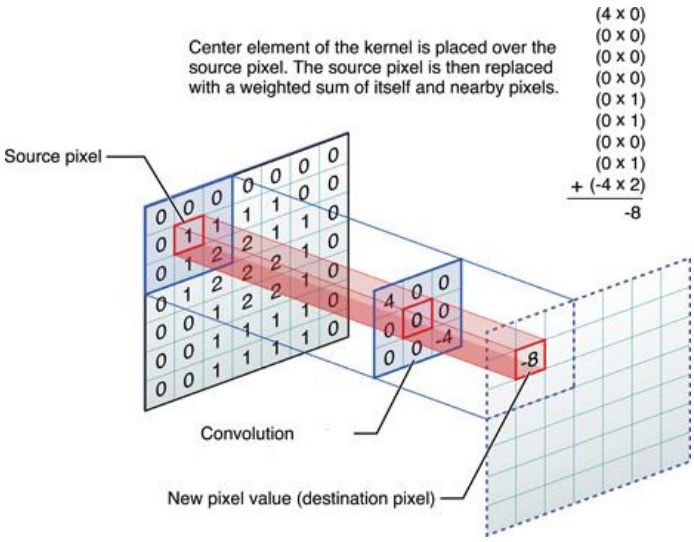


Fig 15. Convolution operation (source)

A filter (usually 3x3 or 5x5) is moved across the image from top to bottom, left to right and each point in the image is convolved with the filter to generate a feature/activation map. This feature map is a representation of features like edges, corners, lines etc.

2) Pooling layer – The main purpose of this layer is to reduce the size of the feature map. The feature map records the exact position of features in the image. What this means is that small changes in the position of the feature in the input image will impact the feature map [21]. We need to make sure the feature detection process is translation invariant. This is where the pooling layers kick in. It reduces the resolution and complexity of the feature map by only outputting the max/average value from a grid [22]. This way, information about the exact position of the maxima is discarded.

3) Fully connected layers – Neurons in this layer have full connections to all activations in the previous layers [23]. Inputs from the previous layers are flattened out and fed to this layer. This layer is usually placed before the output layer.

We will be leveraging the Keras Sequential model to implement a CNN. The first step in this process is to determine the architecture of our model. We will be using the following parameters for the layers (can also be chosen using hyperparameter tuning):

a) 2 convolution layers with 32 and 64 filters in each layer respectively
b) One fully connected layer with 128 units and utilizing rectified linear activation function

This is a summary of our model which has an accuracy of 92% on the train data and 86% on the test data.

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 32)	320
max_pooling2d (MaxPooling2D)	(None, 14, 14, 32)	0
conv2d_1 (Conv2D)	(None, 14, 14, 64)	18496
max_pooling2d_1 (MaxPooling2	(None, 7, 7, 64)	0
flatten (Flatten)	(None, 3136)	0
dense (Dense)	(None, 128)	401536
dense_1 (Dense)	(None, 47)	6063

Total params: 426,415
Trainable params: 426,415
Non-trainable params: 0

Output shape and number of parameters in convolution layer 1

The first layer is a convolution layer. We have used a 3x3 filter size and we have stacked 32 filters. The output of the convolution layer can be computed using this formula [24]:

$$Outputsize(no.ofrows/no.ofcolumns) = \frac{I - F + 2\ x\ P}{S} + 1$$

Where,

- I -> size of input image (28x28)
- F -> size of filter (3x3)
- P -> padding (1)
- S -> strides (1,1)

Substituting the values for these parameters, output size = (28 – 3 + 2/1) + 1 = 28

$$Number\ of\ parameters = (F\ x\ Fx\ number\ of\ channels\ in\ input\ data + bias\ term)\ x\ No.of\ filters$$

Number of channels in the input image is 1 because we use grayscale images. So, the number of parameters = (3x3x1 + 1) x32 = 320.

If you look at the model summary, the input data is in 4D format -> *(None, 28, 28, 32)*. The first argument is the batch size. Since we are not using it, it is set to *None*. The next two arguments are the dimensions of the input image and the last argument corresponds to the depth or the number of filters.

Output shape of max pooling layer 1

Now, this output is passed through a max pooling layer. The output shape of this layer can be calculated using the formula:

$$Outputsize(no.ofrows/no.ofcolumns) = \frac{I - F}{S} + 1$$

We are using a pool size of (2,2). If we have not specified any stride length and it will default to the pool size of 2. So, output size is ((28 – 2)/2) + 1 = 14.

The max pooling layer is just used to down sample the input. So, we don't learn any parameters from this layer.

Output shape and number of parameters in convolution layer 2

The next layer is a convolution layer, and it preserves the size of the input. So, the output of this layer is 14x14x64 because we have chosen to use 64 filters in this layer. Number of parameters we learn in this layer = (3x3x32 +1) x 64 = 18496. Note that, the number of input channels here is 32 because of the input from the previous layer and the number of output channels is 64.

Output shape of max pooling layer 2

This is followed by a max pooling layer and the output size will be ((14 – 2)/2) + 1= 7 and there are no parameters learnt in this layer.

Output shape of flatten layer

The next layer is used to flatted out the output from this layer. So, its output size is 7x7x64 = 3136.

Output shape and number of parameters in dense layer 1

The flatten layer connects to a dense layer and we are using 128 units in this layer. This corresponds to the output shape.

Number of parameters in the dense layer can be calculated by the formula [25]:

$$Number\ of\ parameters\ in\ dense\ layer = (number\ of\ channels\ in\ input\ +\ 1)\ x\ No.of\ units$$

Number of channels passed as input to the layer is 3136 and number of units is 128. So, the number of parameters = ((3136+1) * 128) = 401536

Output shape and number of parameters in dense layer 2/output layer

Similarly, the last dense layer corresponds to the output layer. Since we have 47 classes, its shape is 47. Number of parameters learnt in this layer = ((128 + 1) *47) = 6063.

So, the total number of parameters that this model needs to learn = 320 + 18496 + 401536 + 6063 = 426415.

Let us take a look at what the learnt filters look like. Let's pick the first convolution layer and I am plotting the first 10 filters:

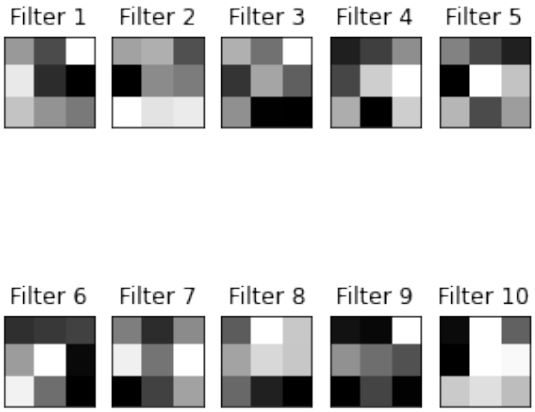


Fig 16. First 10 filters in the convolution layer learnt by the model

These are different 3x3 filters that pick specific features from each image. We can look at incremental stages in this model. Let's begin by looking at the output of applying the first layer to a random training example [26]:

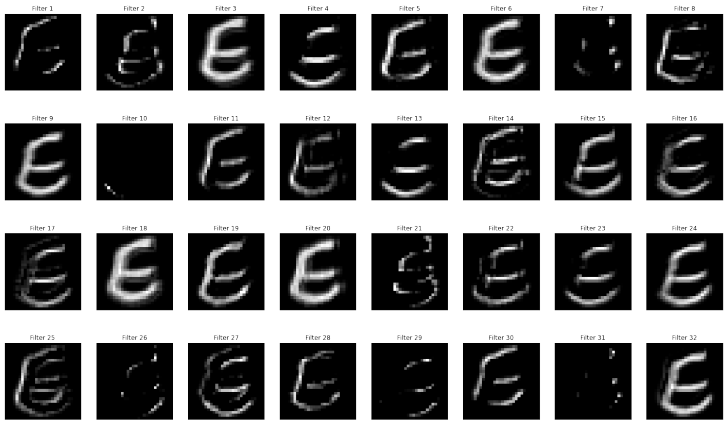


Fig 17. Feature map obtained by applying first convolution layer to a random training example

As you can see, the first layer highlights the lines in the image that delineate the class. The initial layers will learn about low-level features and the subsequent layers will learn a combination of these low-level features. This process continues because we end up stacking layers.

We can confirm this by visualizing the results of the second convolution layer which operates on the inputs from the previous layers. Applying this layer to the same training image, we get:

Fig 18. Feature map obtained by applying second convolution layer to a random training example

This layer captures more general features when compared to the previous layer. At this point it is generally difficult to interpret the actual class/features in the data.

The number of training examples that we have is 112799 and the number of parameters that need to be learnt is 426415 (~ 4 times the number of training images). So, there is definitely going to be some overfitting happening (no of features > no of training data). That is, the network will memorize the training examples. This is evident from the following curves:

Fig 19. Plot of accuracy and error for training and validation data for each epoch

The accuracy improves on the training data for each epoch. For the validation data, it fluctuates. The training error reduces over time. However, the error on the validation set increases.

This is how the model does on our test data:

```
Test loss: 0.5100557804107666
Test accuracy: 0.8584499359130
```

Let's also look at the precision, recall and F1 scores:

label	precision	recall	f1-score	support
0	0.62	0.74	0.68	400
1	0.54	0.73	0.62	400
2	0.87	0.88	0.87	400
3	0.97	0.97	0.97	400
4	0.93	0.89	0.91	400
5	0.87	0.91	0.89	400
6	0.91	0.93	0.92	400
7	0.96	0.98	0.97	400
8	0.88	0.93	0.91	400
9	0.66	0.83	0.74	400
A	0.97	0.91	0.94	400
B	0.97	0.91	0.94	400
C	0.94	0.94	0.94	400
D	0.91	0.89	0.90	400
E	0.95	0.98	0.97	400
F	0.63	0.71	0.67	400
G	0.94	0.92	0.93	400
H	0.95	0.95	0.95	400
I	0.64	0.61	0.62	400
J	0.95	0.89	0.92	400
K	0.96	0.97	0.96	400
L	0.63	0.44	0.52	400
M	0.98	0.97	0.98	400
N	0.91	0.96	0.94	400
O	0.69	0.60	0.64	400
P	0.91	0.97	0.94	400
Q	0.94	0.93	0.93	400
R	0.95	0.94	0.95	400
S	0.89	0.88	0.89	400
T	0.94	0.90	0.92	400
U	0.91	0.92	0.91	400
V	0.92	0.92	0.92	400
W	0.98	0.98	0.98	400
X	0.96	0.94	0.95	400
Y	0.87	0.89	0.88	400
Z	0.90	0.86	0.88	400
a	0.79	0.91	0.85	400
b	0.92	0.91	0.91	400
d	0.94	0.96	0.95	400
e	0.95	0.95	0.95	400
f	0.65	0.48	0.55	400
g	0.64	0.68	0.66	399
h	0.93	0.92	0.92	400
n	0.91	0.91	0.91	400
q	0.71	0.44	0.55	400
r	0.91	0.93	0.92	400
t	0.86	0.90	0.88	400
accuracy			0.86	18799
macro avg	0.86	0.86	0.86	18799
weighted avg	0.86	0.86	0.86	18799

Fig 20. Precision, recall and F1 scores of our model per class and entire test data

Since this is a multiclass classification problem, macro averaging is used to compute these metrics. It reduces the multiclass prediction problem into multiple sets of binary predictions, calculates the corresponding metric for each of the binary cases, and then averages the results together [27].

The weighted macro averages for precision, recall and F1 score are 0.86 each, which is quite good. If you look at some classes like “L”, “f”, “q”, they have a very small recall. Clearly, our model is not doing well on these classes.

What we see from the error plot is overfitting. The model is doing really well on the training data. But it is not able to generalize well. There are many ways to address this:

- Data augmentation – This involves increasing the size of the training data so that it is comparable to the number of parameters to be learnt. This is helpful when you have very few training examples.
- Introducing a dropout layer [28] – In this technique, a bunch of randomly selected neurons are ignored during training. So, their contribution to the activation of downstream neurons is removed and weight updates are not applied during back propagation. This makes the network less sensitive to specific weights and helps is reducing overfitting

We will be employing dropout regularization to our model to fix the overfitting issue:

Model: "sequential_5"

Layer (type)	Output Shape	Param #
conv2d_8 (Conv2D)	(None, 28, 28, 32)	320
max_pooling2d_8 (MaxPooling2	(None, 14, 14, 32)	0
conv2d_9 (Conv2D)	(None, 14, 14, 64)	18496
max_pooling2d_9 (MaxPooling2	(None, 7, 7, 64)	0
flatten_4 (Flatten)	(None, 3136)	0
dense_8 (Dense)	(None, 128)	401536
dropout_2 (Dropout)	(None, 128)	0
dense_9 (Dense)	(None, 47)	6063

Total params: 426,415
Trainable params: 426,415
Non-trainable params: 0

Once this is done, we can see that our model does much better on the validation set:

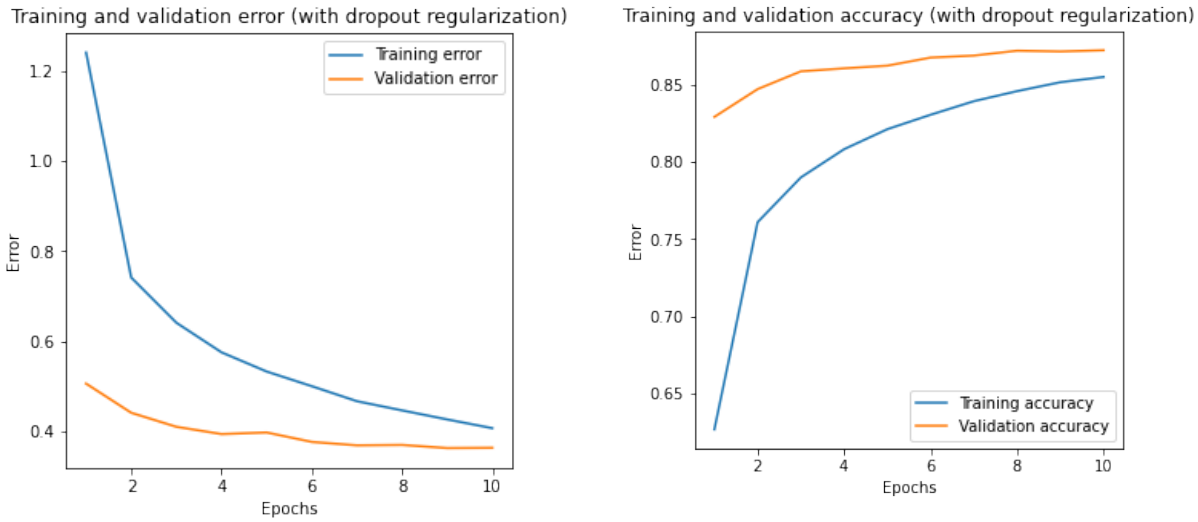


Fig 21. Plot of accuracy and error for training and validation data for each epoch (with dropout regularization)

Finally, let us look at how well our model is doing on the test data:

```
Test loss: 0.37844347953796387
Test accuracy: 0.8710569739341736
```

Let's also look at the precision, recall and F1 scores:

label	precision	recall	f1-score	support
0	0.60	0.82	0.69	400
1	0.53	0.75	0.62	400
2	0.87	0.92	0.90	400
3	0.98	0.99	0.99	400
4	0.91	0.93	0.92	400
5	0.98	0.82	0.89	400
6	0.94	0.91	0.92	400
7	0.96	0.99	0.98	400
8	0.92	0.92	0.92	400
9	0.68	0.84	0.75	400
A	0.93	0.98	0.95	400
B	0.97	0.95	0.96	400
C	0.94	0.96	0.95	400
D	0.94	0.89	0.91	400
E	0.98	0.97	0.98	400
F	0.70	0.55	0.62	400
G	0.91	0.94	0.92	400
H	0.96	0.95	0.96	400
I	0.66	0.65	0.66	400
J	0.93	0.94	0.93	400
K	0.98	0.94	0.96	400
L	0.68	0.44	0.54	400
M	0.96	0.97	0.97	400
N	0.92	0.97	0.95	400
O	0.74	0.55	0.63	400
P	0.95	0.96	0.96	400
Q	0.95	0.91	0.93	400
R	0.96	0.96	0.96	400
S	0.84	0.97	0.90	400
T	0.94	0.89	0.91	400
U	0.93	0.93	0.93	400
V	0.92	0.90	0.91	400
W	0.98	0.99	0.98	400
X	0.98	0.94	0.96	400
Y	0.91	0.88	0.89	400
Z	0.93	0.89	0.91	400
a	0.83	0.91	0.87	400
b	0.93	0.92	0.92	400
d	0.96	0.95	0.96	400
e	0.96	0.94	0.95	400
f	0.62	0.68	0.65	400
g	0.70	0.61	0.65	399
h	0.93	0.94	0.93	400
n	0.92	0.91	0.92	400
q	0.70	0.53	0.60	400
r	0.88	0.94	0.91	400
t	0.85	0.91	0.88	400
accuracy			0.87	18799
macro avg	0.88	0.87	0.87	18799
weighted avg	0.88	0.87	0.87	18799

Fig 22. Precision, recall and F1 scores of our model per class and entire test data (with dropout regularization)

The weighted macro averages for precision, recall and F1 score is 0.88, 0.87 and 0.87 respectively, which has increased from our previous model. The F1 scores for “L”, “f”, “q”, have improved. Our model is still not doing really well for these classes.

We see that the test accuracy has certainly improved with the dropout layer.

Results

We have used the SpinalNet model [29] as our benchmark and my model has an accuracy of 87%, which is not far away from the benchmark.

My model is still doing quite badly on a subset of classes like L, F q, g, 0, O etc.

It would be good to visualize which classes have a high misclassification rate.

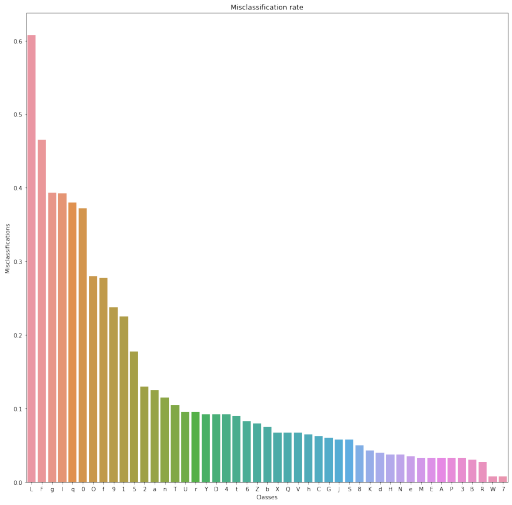
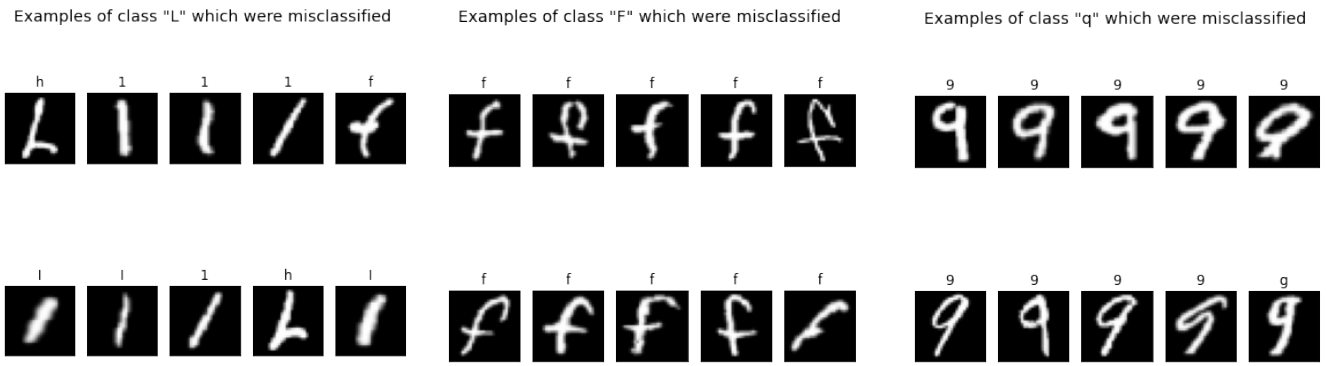


Fig 23. Misclassification rate for test classes

The classes L, F q, g, 0 and O seem to have the highest misclassifications, and this is quite evident from the low recall. Let us plot some test examples in this category to visualize what the data looks like and their predicted labels.



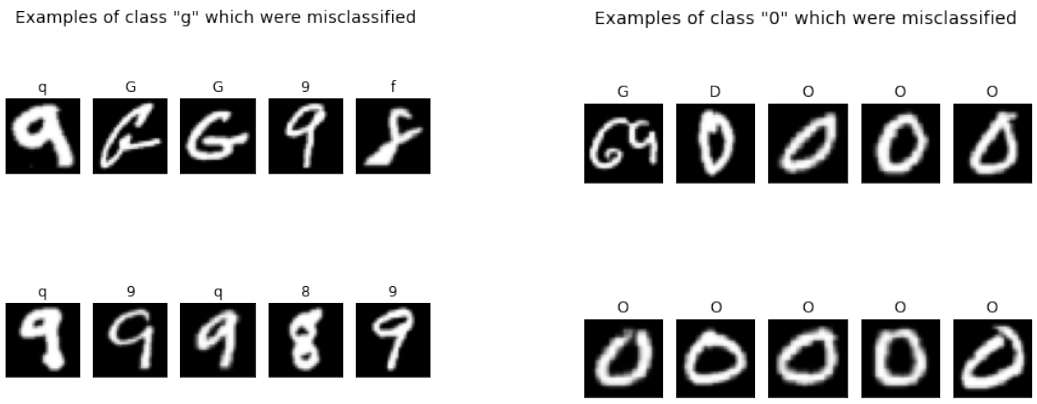


Fig 24. Sample images which get misclassified and their corresponding predicted label

The test images for which we get the prediction wrong are not typical examples and our model seems to be giving a reasonable prediction.

A confusion matrix indicates how well our model has done for all 47 classes:

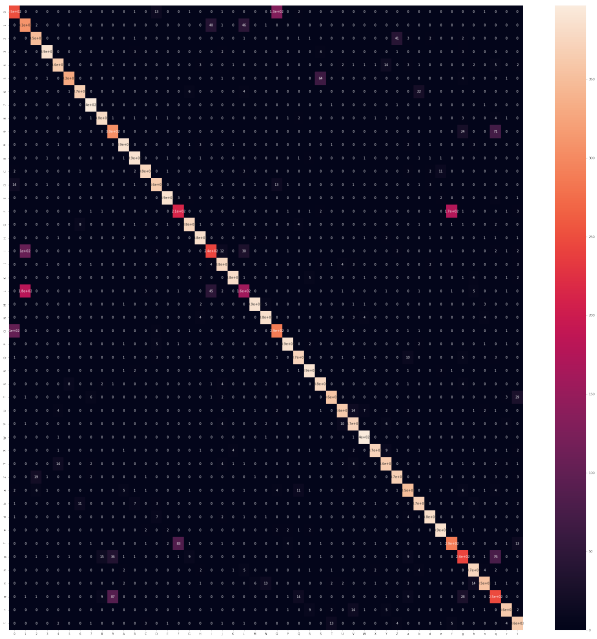


Fig 25. Confusion matrix for model results

The diagonal values are all high and this implies that most classes were identified correctly. There are some high values in the off diagonal elements and that should correspond to images like the ones described in Fig 24.

It would also be a good idea to look at what kind of training data is provided for classes that get confused very often (eg. L and 1 or L and I). The following figure shows a subset of training images of class “L” which are between Q1 and Q3 quartiles described in Fig 6 (with respect to distance from class centroid).

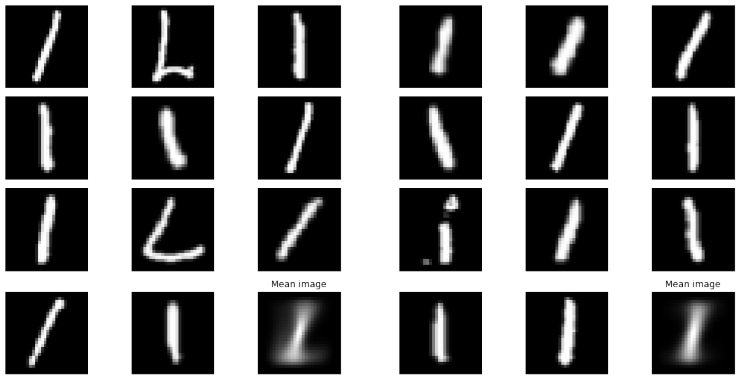


Fig 26. Sample images of class “L” (left) and “I” (right) whose distance from their respective centroid is between Q1 and Q3

From Fig 26, we see that the training images are not very helpful is differentiating these labels. It would be good to filter out such examples from the training data or replace it with mor synthetic meaningful examples.

It is also a good idea to look at the labels associated with the test data. From Fig 24, the images labelled as “F” look very incorrect. If we look at more examples of “F” which we classified incorrectly, this is what we see:

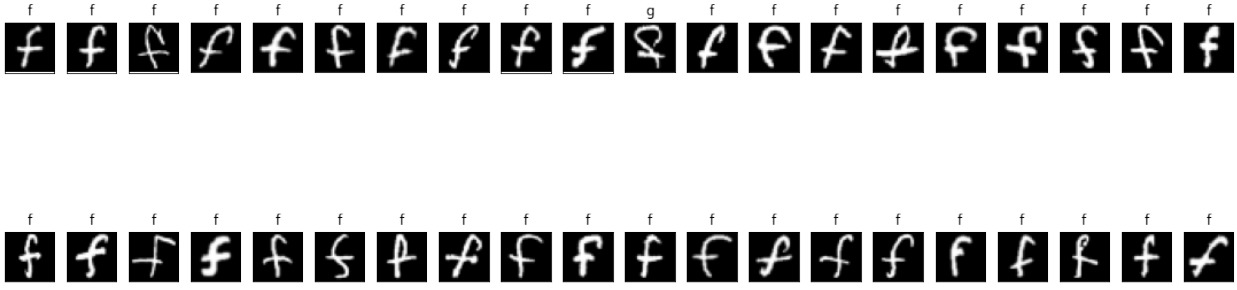


Fig 27. Test data labelled as “F” which were classified incorrectly. Predicted label is on top of each image

They look so much like the lowercase alphabet “f and our model has correctly classified it as “f”! This could in turn be a consequence of bad training examples:

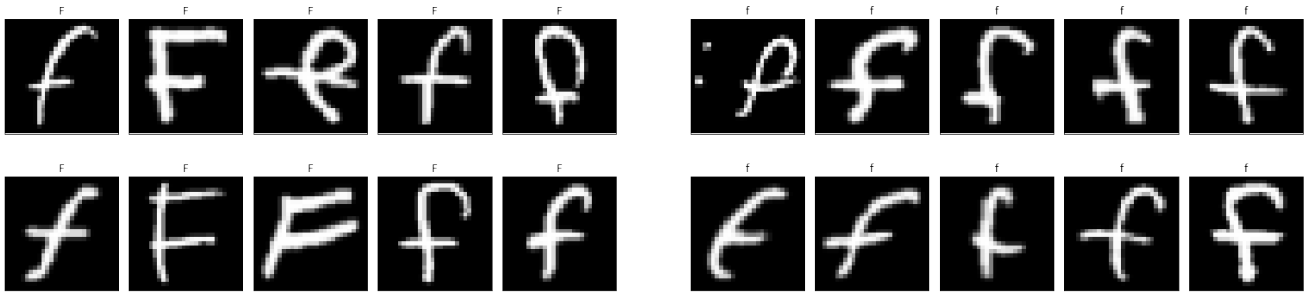


Fig 28. Training data labelled as “F” (left) and “f” (right)

To summarize, there are a number of techniques to improve the accuracy of a model — Data augmentation (when there are fewer training examples), dropout regularization (fix overfitting), Increase the training data size or number of training epochs (fix underfitting), batch normalization (minimize training time) and so on [30]. It is equally important to make sure the training data used on the model is good enough to differentiate one class form the other. The labels associated with the test data should also be valid. If a human cannot accurately determine the class of an image, it is a good indication that the image is not an ideal candidate for training.

Is this model good enough for character recognition?

The short answer is no. Even though we have an accuracy of 87% on the test data, the model does very poorly on some of the classes described in the previous section. This model was not trained on all English alphabets. Moreover, factors like font style, font size and background affect the classification results. To substantiate this, I also tested my model by:

a) Generating new images of text data – Since we are working with numbers and alphabets, it was easy to generate synthetic images, as shown below:

Fig 29. Synthetic images

I tested my model on these new images and the accuracy was about 51%. I also tried translating the characters, switching up the font style and size. The model prediction was impacted by all of these parameters, as shown below:

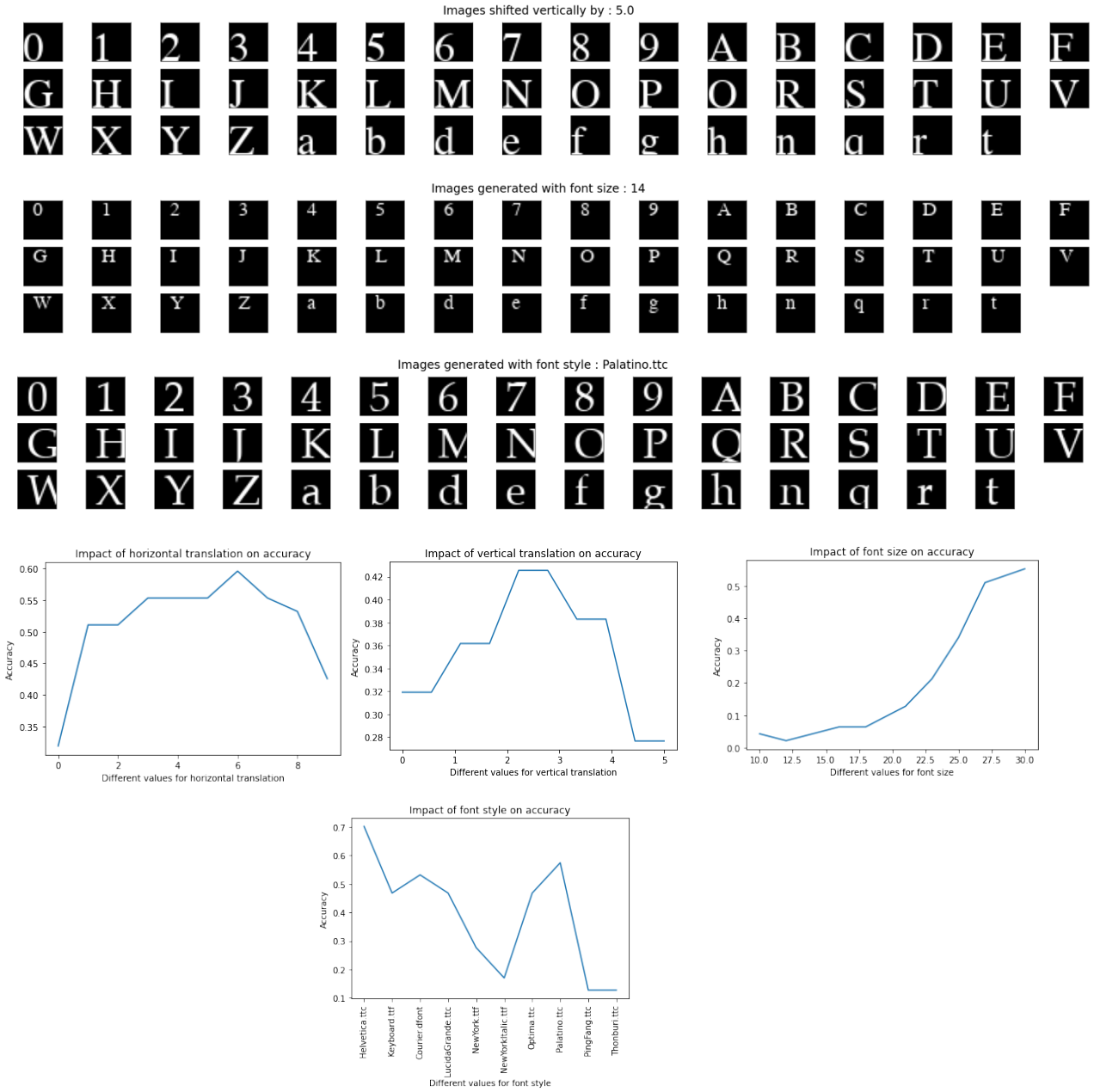


Fig 30. Augmented synthetic images (vertical and horizontal translation) and change in font style and size and its impact on prediction accuracy

Once the new images were generated, I ran it through the model to get the predicted results, to compute the accuracy. What you see here is that the model is senstive to these parameters and this is expected, to some extent. This is because, we have not trained our model to work with these font styles and sizes. If we would like our model to recognize any font style and size, we must train it with additional data.

b) Augmenting existing images – I augmented a couple of classes using an existing training image. I picked a class with very high misclassification rate (“F”) and a class with the lowest misclassification (“3”). The data was augmented by mean centering and performing different translations.

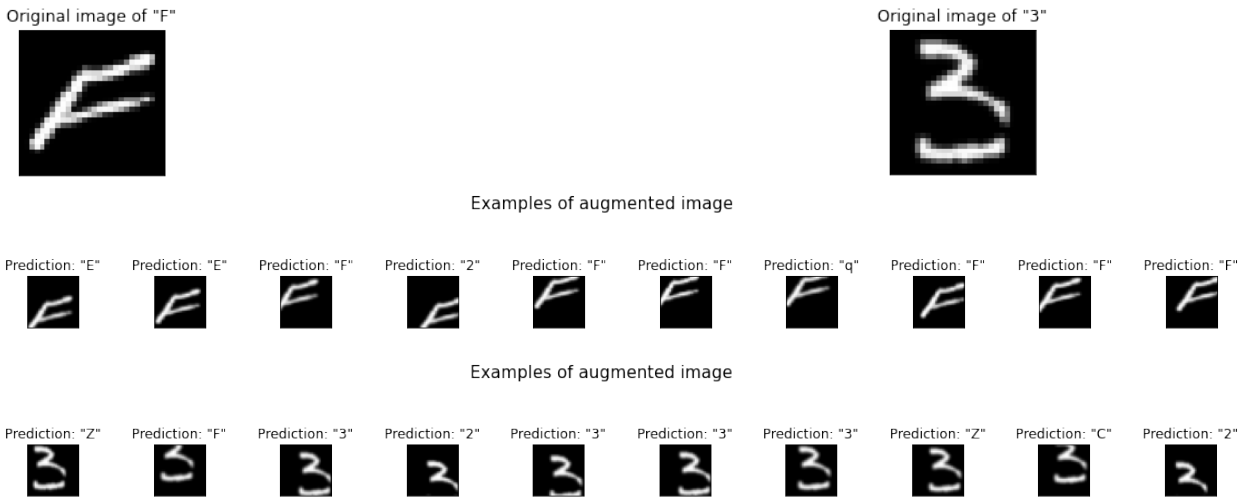


Fig 31. Data augmentation (horizontal and vertical shift) on training image of classes “F” and “3”

I generated around 100 images like these for each of the classes and ran it through the model. For augmented “F”, the accuracy was 0.48 and for augmented “3”, it was 0.2. This is a good indication that model accuracy on the test and validation sets is not sufficient to ensure it generalizes well. We could improve the accuracy of our existing model by using data augmentation.

To summarize, in order to solve the more generic problem of character recognition, we will need to employ data augmentation to have more training examples, train the model on other languages, font styles and sizes, and also identify and correct labeling bias [31].

References

[1] https://www.sciencedirect.com/topics/computer-science/optical-character-recognition
[2] https://www.sciencedirect.com/topics/social-sciences/character-recognition
[3] https://www.digitalnc.org/blog/digital-collections-ocr-what-it-is-and-what-it-isnt/
[4] https://en.wikipedia.org/wiki/Optical_character_recognition
[5] https://research.aimultiple.com/ocr-technology/
[6] https://beebom.com/best-ocr-software/
[7] https://en.wikipedia.org/wiki/Intelligent_character_recognition
[8] https://indatalabs.com/blog/ocr-automate-business-processes
[9] https://www.understood.org/en/school-learning/assistive-technology/assistive-technologies-basics/how-does-optical-character-recognition-help-kids-with-reading-issues
[10] https://www.nist.gov/itl/products-and-services/emnist-dataset
[11] https://www.kaggle.com/crawford/emnist
[12] http://varianceexplained.org/r/digit-eda/
[13] https://www.thoughtco.com/what-is-the-interquartile-range-3126245
[14] https://towardsdatascience.com/explaining-k-means-clustering-5298dc47bad6
[15] https://www.rsipvision.com/image-features-for-classification/#:~:text=Well%20known%20examples%20of%20image,features%20can%20be%20wisely%20selected
[16] https://medium.com/the-owl/multilayer-perceptron-model-vs-cnn-5be5cf87897a
[17] https://www.analyticsvidhya.com/blog/2021/01/image-classification-using-convolutional-neural-networks-a-step-by-step-guide/
[18] https://towardsdatascience.com/simple-introduction-to-convolutional-neural-networks-cdf8d3077bac#:~:text=There%20are%20three%20types%20of,Features%20of%20a%20convolutional%20layer

[19] https://towardsdatascience.com/wtf-is-image-classification-8e78a8235acb
[20] https://www.upgrad.com/blog/basic-cnn-architecture/#:~:text=There%20are%20three%20types%20of,CNN%20architecture%20will%20be%20formed
[21] https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/
[22] https://divsoni2012.medium.com/translation-invariance-in-convolutional-neural-networks-61d9b6fa03df
[23] https://cs231n.github.io/convolutional-networks/#fc
[24] https://medium.com/@kvirajdatt/calculating-output-dimensions-in-a-cnn-for-convolution-and-pooling-layers-with-keras-682960c73870
[25] https://towardsdatascience.com/how-to-calculate-the-number-of-parameters-in-keras-models-710683dae0ca#:~:text=Conv2D%20Layers&text=The%20number%201%20denotes%20the,filter%20that%20we're%20learning.&text=By%20applying%20this%20formula%20to,consistent%20with%20the%20model%20summary
[26] https://machinelearningmastery.com/how-to-visualize-filters-and-feature-maps-in-convolutional-neural-networks/
[27] https://cran.r-project.org/web/packages/yardstick/vignettes/multiclass.html
[28] https://machinelearningmastery.com/dropout-regularization-deep-learning-models-keras/
[29] https://paperswithcode.com/sota/image-classification-on-emnist-letters
[30] https://www.analyticsvidhya.com/blog/2019/11/4-tricks-improve-deep-learning-model-performance/
[31] https://towardsdatascience.com/identifying-and-correcting-label-bias-in-machine-learning-ed177d30349e