



CS464 - Introduction to Machine Learning

Project Progress Report

Handwriting Recognition Using Machine Learning Methods

Asım Güneş Üstüenalp	21602271
Turan Mert Duran	21601418
Mannan Abdul	21801066
Maryam Shadid	21801344
Berdan Akyürek	21600904

1. Introduction

Our lives are becoming increasingly intertwined with the digital world nowadays. And with this development, it is becoming essential that we are able to share information across the physical and digital aspects of our lives. And this necessity extends to ways of transcribing information as well. Our focus in this project is to use machine learning methods to detect one form of transcription, i.e. the handwritten word, and be able to accurately check what that word is. We will use images of handwritten words along with their labels to train our different models. Then we will use our validation dataset to compare the performance of all the models we come up with, going with the model that yields the best results. We then aim to use our test set to check the final metrics of our model to see how it performs.

Our dataset contains 330,961 training images, 41,370 validation images and 41,370 test images. The dataset has 3 CSV files that contain the URL of the images in the dataset along with their labels. This dataset is the same as the one we have mentioned in our proposal and we may choose to incorporate additional datasets in the latter stages of the project for better results.

2. Work Done Until Now

Until the date this report is written, the dataset is preprocessed and a simple model initialized.

2.1. Preprocessing

Before starting the training process, the dataset is examined. It is realized that data is not standard and some preprocessing may be necessary. An example data that needs preprocessing is as shown below.



The problem is, this photo is labeled as “MARTIN” but it contains the word “PRENOM” too. The data contains such photos in a great amount. So the data needed to be cropped to the necessary content. So, each image is cropped similar to the image below.



In order to do this, Google Vision API is used[]. This API simply detects each word in the image separately and gives coordinates of it in the image. We created .cropinfo files that contain information returned by Google API without doing actual cropping.

For example, .cropinfo file for the uncropped image above named TEST_0006.cropinfo is as follows:

```
"PRENOM : MARTIN" (5,-1) (184,-1) (184,19) (5,19)
"PRENOM" (5,0) (56,0) (56,18) (5,18)
": " (61,0) (70,0) (70,19) (61,19)
```

```
"MARTIN" (83,-1) (184,0) (184,19) (83,18)
```

As it can be seen, each word is detected separately. The four coordinates following each word are the four corners of the rectangle covering that word. Using these coordinates, it is possible to get rid of unnecessary words and non-text areas. This way, data is converted to more clean data and total data size is reduced. The function implementation that takes a path for an image and returns the .cropinfo file is as follows:

```
def detect_text(path):
    """Detects text in the file."""
    from google.cloud import vision
    import io
    client = vision.ImageAnnotatorClient()

    with io.open(path, 'rb') as image_file:
        content = image_file.read()

    image = vision.Image(content=content)

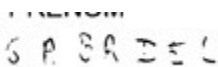
    response = client.text_detection(image=image)
    texts = response.text_annotations
    str_to_ret = ""
    for text in texts:
        str_to_ret +=
            "{}".format(text.description.strip().strip("\n")).strip() + " "

        vertices = (['({},{})'.format(vertex.x, vertex.y)
                     for vertex in text.bounding_poly.vertices])

        str_to_ret += '{}\n'.format(' '.join(vertices))

    if response.error.message:
        raise Exception(
            '{}\nFor more info on error messages, check: '
            'https://cloud.google.com/apis/design/errors'.format(
                response.error.message))
    return str_to_ret.strip("\n")
```

This function is derived from the example function provided in the API documentation[https://cloud.google.com/vision/docs/ocr#vision_text_detection-python]. By examining the cropinfos, it is realized that there are faulty results returned by Google API. Some text is undetected or wrong detected by Google API. For example, for the image below, no text is detected by Google.



SERIAL

For this situation, when no text is found, it is possible to leave it as it is without preprocessing or extracting from the dataset.

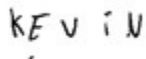
Also, for the image below, the text is detected as “BALTHA2AR” despite it is “BALTHAZAR”:



NOM
BALTHAZAR

For this situation, comparison can be done in terms of approximate string matching. Instead of exact detection of a word, a similar word also can be interpreted as correct.

Lastly, the following image is detected as three different words as “KE”, “V” and “IN” although it is a single word “KEVIN”:



NOM
KEVIN

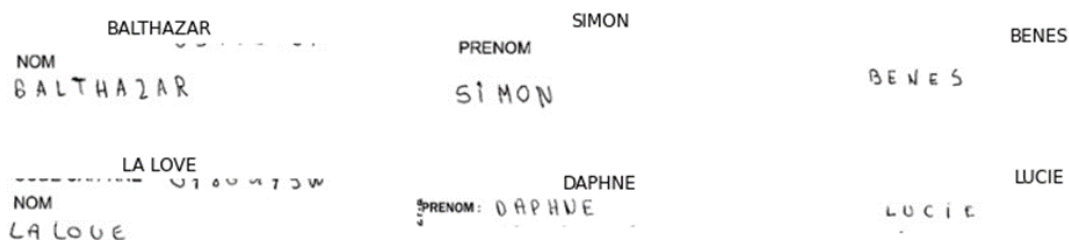
For this situation, combinations of consecutive words should also be checked.

Actual cropping is not done yet. It will be done according to choices explained above in the future.

2.2. Initial Model

While building our initial model, preprocessing was kept to a minimum so we could focus on building a working model first and get its performance measure. We would then move on to exploring all the different options we have in terms of preprocessing our dataset after a discussion with our TA, this would also help us see how effective our preprocessing was in terms of increasing the performance of our models.

To get started, we first use the pandas python library to read the CSV files containing the URLs of our images and their labels. We show a small subset of these images in a plot to be able to see what the dataset looks like. The plot is shown below:



BALTHAZAR
NOM
BALTHAZAR

SIMON
PRENOM
SIMON

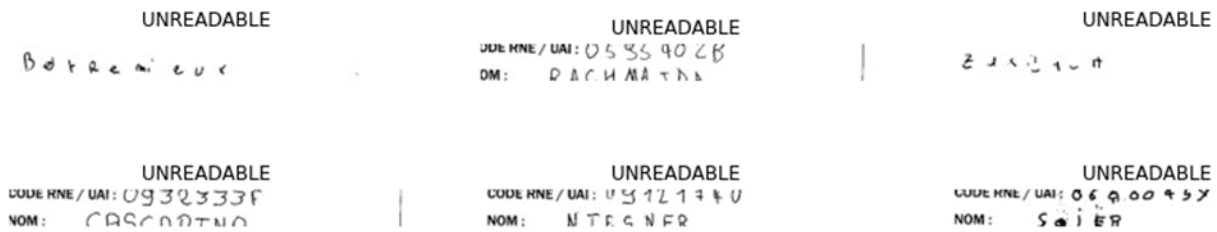
BENES

LA LOVE
NOM
LA LOVE

DAPHNE
PRENOM: DAPHNE

LUCIE

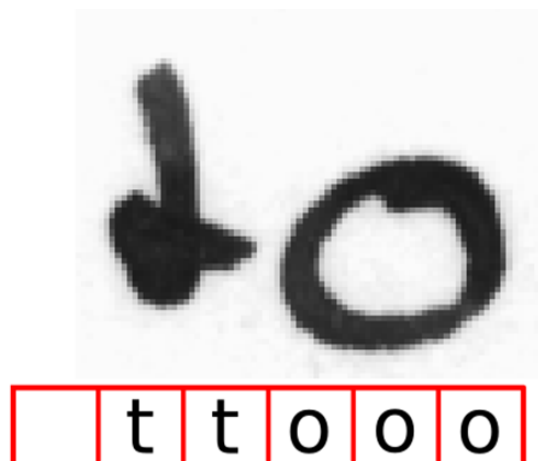
The first step in preprocessing our images was to remove all such images whose labels were null. This step removed 565 images in the training set and 78 images in the validation set. Looking through the dataset, we also found some images that were labeled unreadable, so the next step was to remove these images from the dataset. These images were either cropped such that no words could be seen or were just illegible. Some example images are shown below:



After removing the unreadable images, all the labels were set to uppercase as we had some labels that were in lowercase and we wanted uniformity in our labels.

After cleaning up the dataset, we read all the images into an array. For each image, we resize the image to width = 256 and height = 64 and then normalize the image values to be between 0 and 1. Resizing each image can also help us remove some of the unwanted text that is present in each image but the method is somewhat crude and not accurate so we will look for alternatives to cleaning up the images in the latter stages of the project. Each image is read using the cv2.imread method from the cv2 library as a grayscale image. We read all the images in the training set using the procedure mentioned above into an array that has the training images and the validation images using the same procedure into an array that contains validation images. After the images are read, we reshape the image arrays so they can be fed into the CNN. At the moment, we are not using the whole dataset but only 30,000 training images and 3,000 validation images. The reason is that when we reshape the training array into a format we can input to the CNN, the training images array takes on the shape of (num_images, 256, 64, 1). If num_images is the whole of the training dataset, the array needs a whopping 40.4 GiB to be stored in memory. In the latter parts of the project, we will find ways to be able to use the whole of the training and validation dataset to be able to get better performance metrics from our models.

The next step is to prepare our labels for Connectionist Temporal Classification (CTC) loss. The problem with using neural networks for text recognition is that we would first need to annotate our dataset at a character level. We also need to divide the image into let's say 10 horizontal positions and for each of those positions annotate on an image which character is in that horizontal position. An example is shown below:



Our neural network would then be trained to output a character score for each of these horizontal positions and the character with the highest score would be chosen for that position. But there are several problems with this approach. It is very time-consuming to annotate each image in the dataset on a character-level and we would still need to find a way to convert the character scores we get from our neural network into the actual character. Another issue that may arise is since a single character can span multiple horizontal positions, we might get repeated characters as shown above. If we try to resolve this by removing all repeating characters, we run into another issue: if our word was 'too', removing all repeating characters would yield the wrong result. Using CTC helps us because we only need to tell the CTC loss function the result of the text that appears in the image without any concern for the width of the characters nor do we need to worry about their positions. The CTC loss function helps us guide the training of our neural network as we feed it the output matrix of the NN and the corresponding ground-truth label. The loss function tries all the possible alignments of the ground-truth label text in the image to check which alignment is most possible. We train our NN with the goal of minimizing our CTC loss which would help us get a more accurate model. More information on CTC loss is available here:

<https://towardsdatascience.com/intuitively-understanding-connectionist-temporal-classification-3797e43a86c>

So, we prepare our labels by converting them into numbers which represent the position of the character in our alphabet. For the training set, we initialize an array and pad it with -1s. Then for each label, we convert the labels into numbers and add them to the aforementioned array. An example is shown below:

```
True label : NOUR
train_y : [13. 14. 20. 17. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.
-1. -1. -1. -1. -1. -1.]
train_label_len : [4.]
train_input_len : [62.]
```

After doing this for each label in both the training and validation sets, we start to build our model.

We use a Conv2D layer from `tf.keras.layers` as our first layer, with 32 filters and a kernel size of (3, 3). We then use the BatchNormalization function to normalize our values as it is supposed to help the learning process run faster. We use a relu activation function with our convolving layer and MaxPooling2D function that has a (2, 2) pooling window which picks the highest value in that window and replaces the whole window with it. This helps us remove features in the image that are not useful and it will also remove the chances of the model overfitting as the model will not try to learn every single parameter. Two more convolving layers are added after with a dropout of 0.3 to remove the effects of overfitting further.

Our model here is still not complete. If we go back to our preprocessing steps, we did not perform any type of image segmentation, which means we could not classify the characters individually and piece them together to form our words. For this reason, after adding our

convolving layers to our model, we add a layer to reshape the output from the previous layer and add a dense layer with 64 units and a relu activation function which means each of these 64 units are deeply connected (to every neuron of the preceding layer). So the dense layer can be thought of as consolidating the output of the CNN part of our model. We then use a Long Short-Term Memory (LSTM) layer for the RNN part of our neural network. Research indicated that using a bidirectional LSTM layer can lead to better learning so that is what we are using because by doing this, our output layer can get information from both the past and future states. This is because the layer trains the model once on the input sequence as is and once on the input sequence reversed. RNNs are good for training models where sequences matter, so applications like speech recognition, etc. As mentioned before, we need this property because we have not segmented our images into characters so we cannot just classify based on that. After adding the RNN layers, we again use a dense layer with `num_of_chars` (in our alphabet) + 1 (for ctc blank) (total = 30) units with a softmax activation function. The output shape of our predictions is (64, 30) which means that the model will predict words of length 64 (to allow for enough ctc blanks) and will have the probability of each of the 30 alphabets that we defined earlier.

We also use 2 models here. `Model_final` is the same as the model defined above except the output is connected to the ctc loss function that we define which uses `ctc_batch_cost` from Keras to calculate the cost. We use the function `model_final.fit()` to train this model with the objective of minimizing the ctc loss. The other model named 'model' then uses the weights from `model_final` and we then use this model to make our predictions. Training the model took approximately 12 hours and over 60 epochs, we got the loss down from 20.8 to 2.1.

Note that we still need to decode the output of the NN, and for that we use the `ctc_decode` function's output, convert it back into a word and we have our prediction. For our model, we achieved an 87.77% character accuracy but only a 73.63% word accuracy. In the latter stages of the project, we plan to devise ways to improve this accuracy and come up with other models to see if they can perform better.

3. Work To Be Done

Work will be done to check for ways in which we can utilize all of our dataset and for ways to better preprocess it so we can get better results from our models. Preliminary research also showed us that there are 2 other innovative methods we may be able to use for our application purposes. The first one is to use FasterRCNN as it is suitable when you have many objects (characters) in one image. The second one is transfer learning, using existing learning to help train models in similar situations. One option here is to use ResNet50 which has been trained on more than 1 million images already. But further research is needed to see if these options are viable or not for handwriting recognition – All Group Members

Fine-tune our existing model to perform better. One problem was that this model was not trained on a GPU and it took a long time because of that, the reason being that we were getting a GPU

out of memory error. The research will be done on how to train the model in batches without having to load all the data in the RAM at once, we will be able to use all of our training and validation data to get better performance from the model as we were only able to use less than 10% in this iteration. The results will be compared against other models to see which one performs best. – Mannan Abdul

Evaluate the dataset using a PCA and SVM approach. Upon researching, we found that SVM models give highly accurate results for handwritten digit recognition. To test this on our dataset for handwriting recognition, we would have to segment each word into individual characters. The Principal Component Analysis algorithm will then be used to reduce the dimensions of the preprocessed images. After this, an SVM model will be used for classification – using Sklearn libraries to train the preprocessed images. After classification, performance metrics will be used to compare accuracy against other models, and the one with the highest accuracy will be determined. – Maryam Shahid

Letter segmentation will be implemented. Letters can be separated from each other. This way it is possible to classify each letter separately among 26 letters. This way it will be possible to train classification models like kNN and Naive Bayes. - Berdan Akyürek

By using the separated letters that Berdan Akyürek is going to get, I am going to evaluate the dataset by using the kNN approach. According to the research that we have done so far, it gives the second-highest accurate results for digit recognitions. I am going to train the model using the kNN algorithm using separated letters and by using our preprocessed datasets and unprocessed datasets. I am going to choose the dataset with the highest accuracy. - Turan Mert Duran

As a group, we will train different models on our dataset and choose the most performative one as our final model. In our research, we found that the Gaussian Naive Bayes algorithm performs reasonably well for recognizing hand-written text. I will train a model using the Gaussian Naive Bayes algorithm using the segmented letters that will be supplied by Berdan's work. I will train the model using both preprocessed and unprocessed datasets and choose the best result. - Asim Gunes Ustunalp