2022

INTRODUCTION TO DEEP LEARNING PROJECT- MEDIPOL UNIVERSITY

HANDWRITING RECOGNITION USING OPENCV AND TENSORFLOW

EMRE ALPÖĞÜNÇ 63190011 ,ONUR TOSUN 63180072

ONUR TOSUN 63180072 EMRE ALPOGUNC 63190011

ABSTRACT:

For this project, we wanted to somehow find a way to turn a handwritten sentence or a word to a text on computer. The fascinating part of this was that, how can we let the machine know that what are the words. So basically, the process that we imagined was that we were going to create a model so that when we give it a handwritten sentence or any sample, it will display us the exact handwritten sample on to the screen. If we could achieve this somehow, we were also planning to translate the written part to a given language to act as a converter. A similar example can be the Google's translate function using camera. Even though it's doing it in real time, this was a similar concept with few compromises from a user's quality of life standpoint. We first tried to find a way to detect the letters of a given sample. There were already established papers and articles as well as examples.

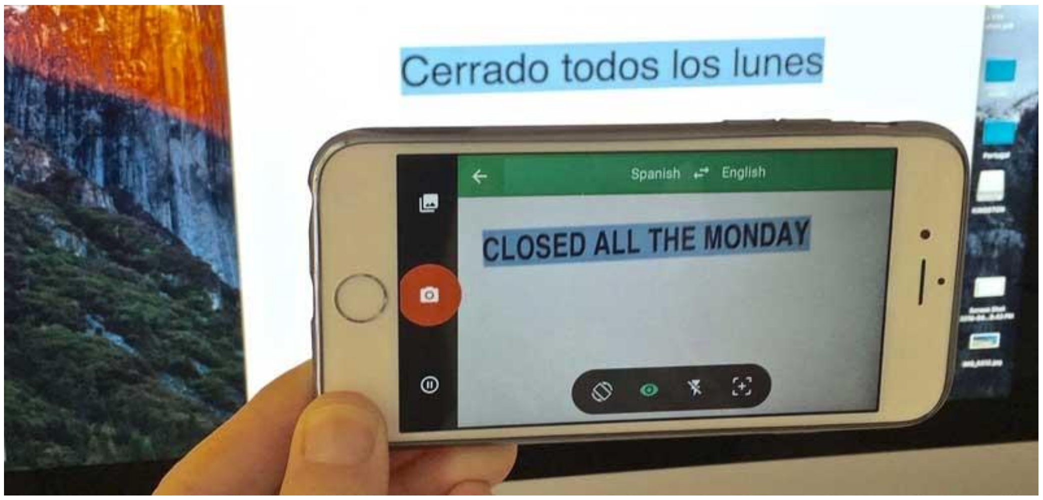
We have two methods to solve this problem, and, in this report, we are also trying to decide to which way is working better. These methods are going to be explained in more detail within the later pages (also the comparison as which one is better for constant use). To explain it briefly, we tried different ways to calculate the length differences between two label boxes (Which we receive as outputs from OpenCV, more on that later). In the end, both models can give us proper results within various circumstances. We tried different instances and different platforms to write (Microsoft Paint, Pen & Paper etc.) And we believe that both perform sufficient for our needs but of course after some comparations we decided to go with the first model.

INTRODUCTION:

In this day and age, communicating with one another is easier than ever. But this said, there are some instances where we become choiceless. English is the most common language in the world. This said, we see tons of instances where a person in particular struggles to understand a certain sentence or a specific word since they don't know the language that much. We wanted to somewhat eliminate that obstacle with our project. But not only people who are using this will benefit from this, we as a team will also learn a lot of things about the marvels of deep learning.

How can computer scan and comprehend the letters that are given in a handwritten sample, how can we tell the computer where a word ends the next one starts in a sentence? Especially the latter question was difficult for us to answer since we haven't worked on a project like this. There are much easier and possibly more refined models that was done by professionals but with the way we did, we were able to understand the fundamentals of how we process these words. We built and tinkered different features and properties and as we build and continue to increase the complexity of this model, we slowly started understanding more and more. There is of course a room for improvement but as far as we are concerned, we came a long way from where we were once. Our model can detect the letters individually, can calculate the distances between them and distinguish them as where a word starts and ends and at the end It writes both the word and sentence as well as the translated version as outputs. We only used database for the letter detection since the challenging part is to how to proceed from there. As we have mentioned within the abstract, there two approaches we took to solve this problem. Our results are correct in most instances, but we are going to share both of our successful and unsuccessful instances to show how our model(s) perform in a daily use.

The process we went through was explained as topic by topic. We hope that we can share every little detail as thoroughly as possible. If wanted, from the sections page the specific part can be visited.



RELATED WORK:

As related work, we wanted to use something that can be helpful to understand the fundamentals of the program and the project itself. In order to do that, we had to choose a program and project that would help us quite a lot with our first part of the program. Which is the letter and as well as the handwriting recognition. For that, we went with:

Robust Handwritten Text Recognition in Scarce Labeling Scenarios: Disentanglement, Adaptation and Generation by Lei Kang.

We especially tried to use this material since it was explaining everything in a tremendous detail which was what we needed for this project.

The way we differ from this, and many more similar approaches is that for our references, there was no example of detecting the words without any libraries. Every one of the references that we tried to get help with never tried to take an approach where the words can be detected without any library. This was the crucial part that we wanted to understand from this project as well. There are toolkits like Tesseract for doing what we do quite easily, but we wanted to do the word detection and translation without any additional hep with datasets. We tried mathematical approaches to detect the words instead of teaching the model the words themselves. The computer might not know the results completely, but this will show that it’s possible to accomplish without any additional libraries.

DATA:

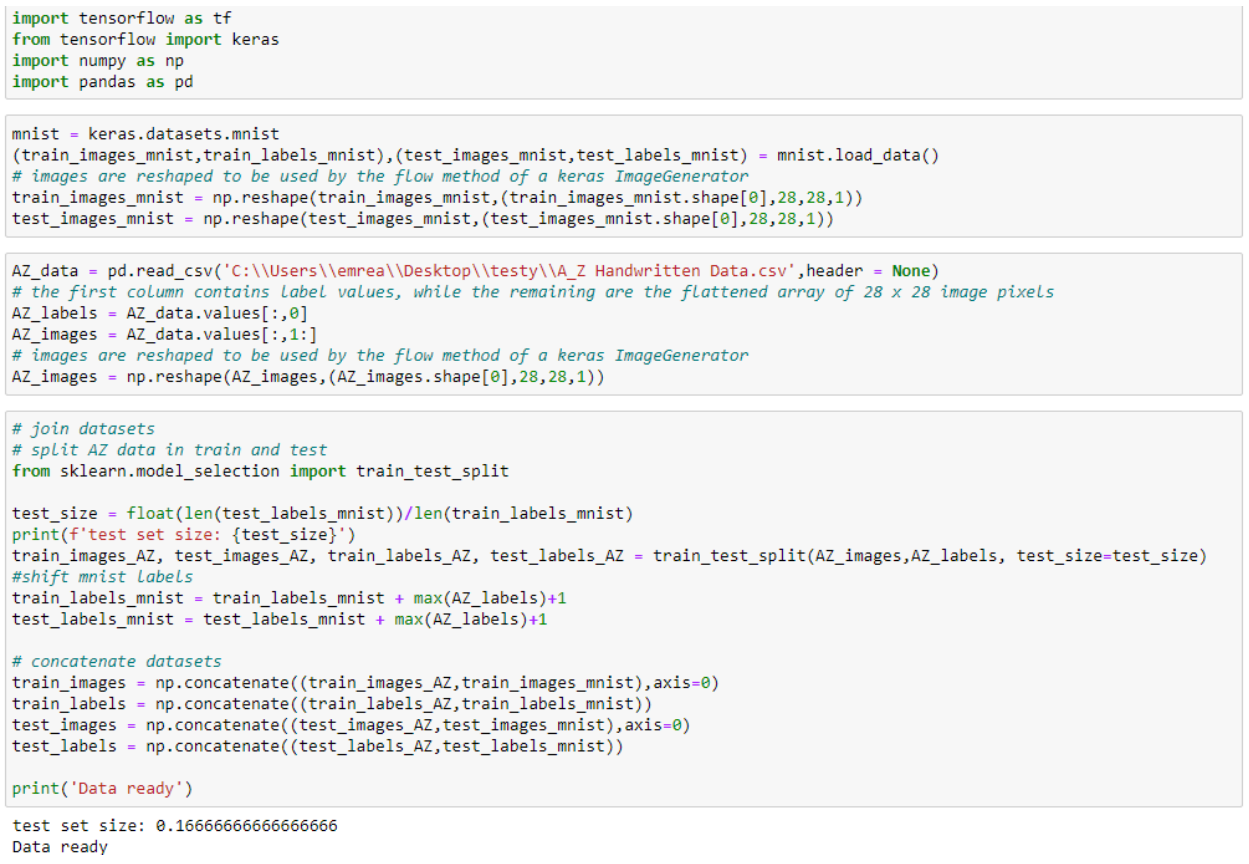
The datasets that we use are quite popular and accessible. For digit recognition, we used no dataset other than MNIST. For the handwritten data recognition, we used the Kaggle A to Z dataset. For handwriting samples, we wrote them ourselves to see how they were performing. To explain the datasets briefly, MNIST has 60,000 training examples and 10,000 test samples. They are all written in handwritings, and we’ve worked with this dataset for multiple times, so we thought that why not use it again.

Kaggle A to Z on the other hand will help us with the letters that are handwritten. All of the letters are in grayscale and there are total of 370,000+ English alphabets within the dataset itself. The operations we apply can be seen within the methods section.

ONUR TOSUN 63180072 EMRE ALPOGUNC 63190011

METHODS:

Right here, we prepare the data and create the train and test portions accordingly. We load the datasets and import the first fundamental libraries:



```
import tensorflow as tf
from tensorflow import keras
import numpy as np
import pandas as pd
```

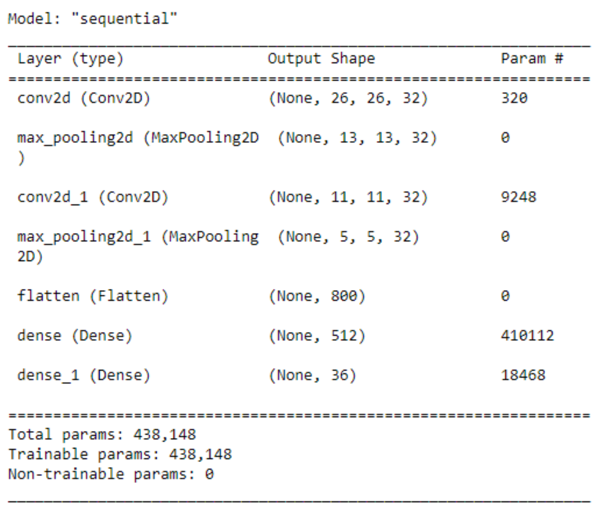
```
mnist = keras.datasets.mnist
(train_images_mnist,train_labels_mnist),(test_images_mnist,test_labels_mnist) = mnist.load_data()
# images are reshaped to be used by the flow method of a keras ImageGenerator
train_images_mnist = np.reshape(train_images_mnist,(train_images_mnist.shape[0],28,28,1))
test_images_mnist = np.reshape(test_images_mnist,(test_images_mnist.shape[0],28,28,1))
```

```
AZ_data = pd.read_csv('C:\\Users\\emrea\\Desktop\\testy\\A_Z Handwritten Data.csv',header = None)
# the first column contains label values, while the remaining are the flattened array of 28 x 28 image pixels
AZ_labels = AZ_data.values[:,0]
AZ_images = AZ_data.values[:,1:]
# images are reshaped to be used by the flow method of a keras ImageGenerator
AZ_images = np.reshape(AZ_images,(AZ_images.shape[0],28,28,1))
```

```
# join datasets
# split AZ data in train and test
from sklearn.model_selection import train_test_split

test_size = float(len(test_labels_mnist))/len(train_labels_mnist)
print(f'test set size: {test_size}')
train_images_AZ, test_images_AZ, train_labels_AZ, test_labels_AZ = train_test_split(AZ_images,AZ_labels, test_size=test_size)
#shift mnist labels
train_labels_mnist = train_labels_mnist + max(AZ_labels)+1
test_labels_mnist = test_labels_mnist + max(AZ_labels)+1

# concatenate datasets
train_images = np.concatenate((train_images_AZ,train_images_mnist),axis=0)
train_labels = np.concatenate((train_labels_AZ,train_labels_mnist))
test_images = np.concatenate((test_images_AZ,test_images_mnist),axis=0)
test_labels = np.concatenate((test_labels_AZ,test_labels_mnist))

print('Data ready')
```

```
test set size: 0.16666666666666666
Data ready
```

First and foremost, we wanted to finally be able to do the letter recognition. To do that, we first loaded our data, split into chunks of train and test data for both databases. As we've said, for the integers we used the MNIST dataset which is crucial to our operation as well as the Kaggle A TO Z dataset. After that, we created our model. Our model includes the following as the layers and other adjustments:

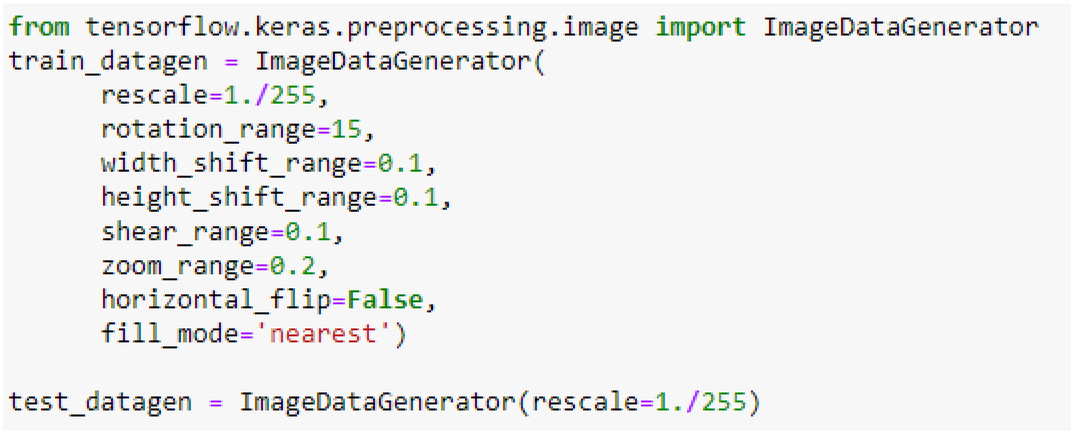


```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2D  (None, 13, 13, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 11, 11, 32)        9248

 max_pooling2d_1 (MaxPooling  (None, 5, 5, 32)         0
 2D)

 flatten (Flatten)           (None, 800)               0

 dense (Dense)               (None, 512)               410112

 dense_1 (Dense)             (None, 36)                18468

=================================================================
Total params: 438,148
Trainable params: 438,148
Non-trainable params: 0
_________________________________________________________________
```

- The first layer is a convolutional layer (Conv2D) with activation function of “ReLU”. The kernel size is (3,3) and filters were set as 32. The input shape is as the data section suggested is 28x28 so went with (28,28,1).
- After that we went with a Maxpooling(MaxPooling2D). The pool size is (2,2).
- For the second Conv2D layer, we wanted the same properties as it was also working accurately in our other instances. Thus, convolutional layer (Conv2D) with activation function of “ReLU”. The kernel size is (3,3) and filters were set as 32.
- After that we again went with a Maxpooling (MaxPooling2D). The pool size is (2,2).
- After all of this was done, we applied a Flatten operation to reduce the shape of the model.
- We used a Dense layer as well after this which had 512 units and activation function of “ReLU”.
- Lastly, we had to use an Output layer and for that, we used the length of the **train_labels** after applying a **np.unique** method to filet out the copies.

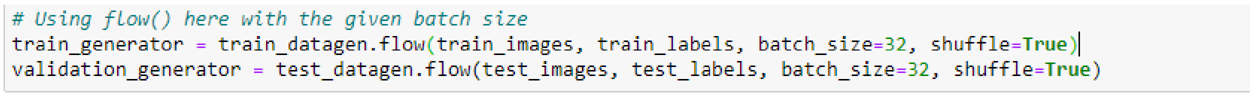
As for the compiling, we used RMS Prop, and we are also going to try with different optimizers as well, but this was the one we chose to put of all after the experiments. The learning rate is 0.0001 and loss was determined as “sparse categorical crossentropy”.

We then tried to replicate the tensorflows’s keras image processing tool ImageDataGenerator. We trained the Data Generator with these properties:



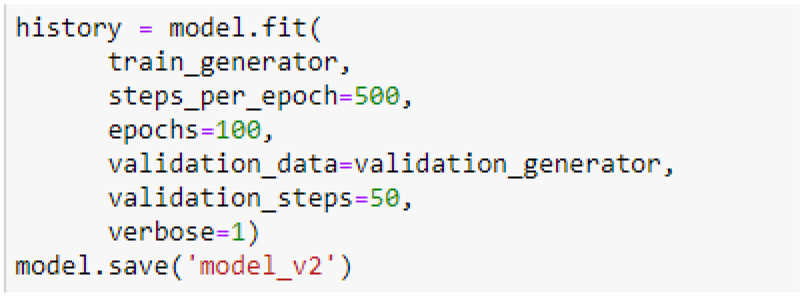
```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
train_datagen = ImageDataGenerator(
      rescale=1./255,
      rotation_range=15,
      width_shift_range=0.1,
      height_shift_range=0.1,
      shear_range=0.1,
      zoom_range=0.2,
      horizontal_flip=False,
      fill_mode='nearest')

test_datagen = ImageDataGenerator(rescale=1./255)
```

After doing this part ads well, we went with the flow() function of the ImageDataGenerator. This helps us to perform image augmentation while we are simultaneously training the model itself.

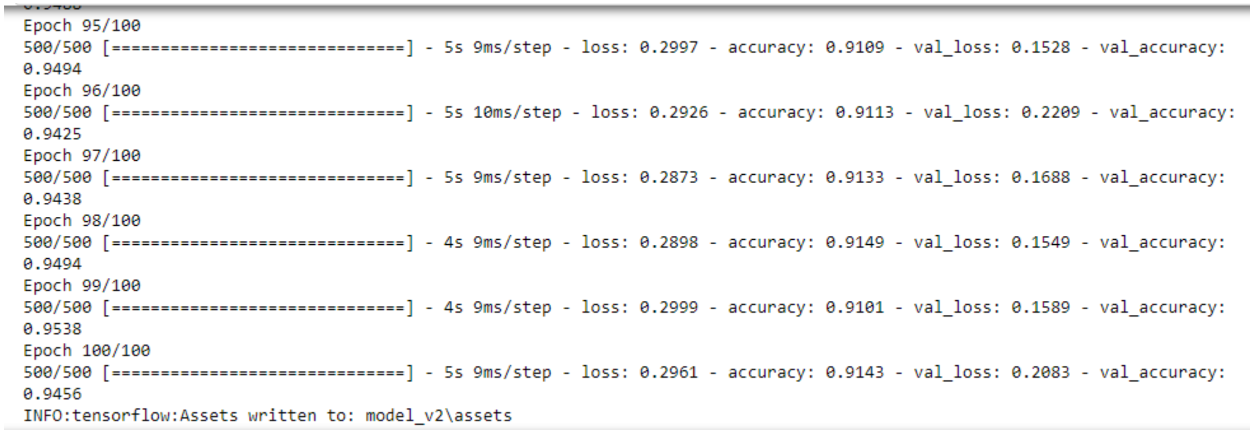


```
# Using flow() here with the given batch size
train_generator = train_datagen.flow(train_images, train_labels, batch_size=32, shuffle=True)
validation_generator = test_datagen.flow(test_images, test_labels, batch_size=32, shuffle=True)
```

After that we do the fitting operation and here are the properties:

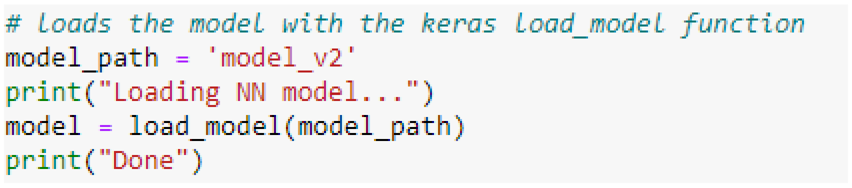


```
history = model.fit(
      train_generator,
      steps_per_epoch=500,
      epochs=100,
      validation_data=validation_generator,
      validation_steps=50,
      verbose=1)
model.save('model_v2')
```



```
Epoch 95/100
500/500 [==============================] - 5s 9ms/step - loss: 0.2997 - accuracy: 0.9109 - val_loss: 0.1528 - val_accuracy:
0.9494
Epoch 96/100
500/500 [==============================] - 5s 10ms/step - loss: 0.2926 - accuracy: 0.9113 - val_loss: 0.2209 - val_accuracy:
0.9425
Epoch 97/100
500/500 [==============================] - 5s 9ms/step - loss: 0.2873 - accuracy: 0.9133 - val_loss: 0.1688 - val_accuracy:
0.9438
Epoch 98/100
500/500 [==============================] - 4s 9ms/step - loss: 0.2898 - accuracy: 0.9149 - val_loss: 0.1549 - val_accuracy:
0.9494
Epoch 99/100
500/500 [==============================] - 4s 9ms/step - loss: 0.2999 - accuracy: 0.9101 - val_loss: 0.1589 - val_accuracy:
0.9538
Epoch 100/100
500/500 [==============================] - 5s 9ms/step - loss: 0.2961 - accuracy: 0.9143 - val_loss: 0.2083 - val_accuracy:
0.9456
INFO:tensorflow:Assets written to: model_v2\assets
```

After this, we lead the model with the specified model path.

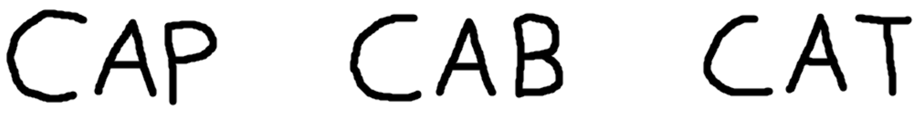


```
# loads the model with the keras load_model function
model_path = 'model_v2'
print("Loading NN model...")
model = load_model(model_path)
print("Done")
```

We try to make some adjustments to the photo after we import it. We are using OpenCV for this part additional to what we already have.

First, we print the original image on a graph using the matplotlib library, after that we grayscale the image using the OpenCV tools, and we crop it to the wanted size to erase any unnecessary parts and lastly, we blur the image (Using the Gaussian Blur) to make it more easy to read. For our sample photos especially, we didn't need to make too many crop operations since it was not really needed but if we needed to do so we can here from the highlighted section of the code which will be shown here:

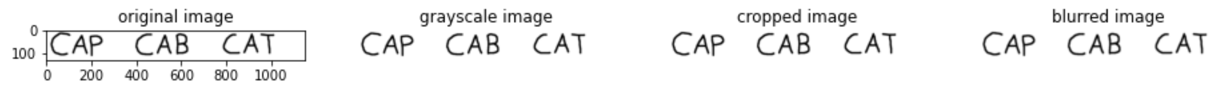
Our Sample Image for demonstration:



As a disclaimer, the image after each operation was titled so if needed it can be checked out from there.

```
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
cropped = gray[2:,:]#Crpping's done here.
blurred = cv2.GaussianBlur(cropped, (5, 5), 0)


%matplotlib inline
from matplotlib import cm
fig = plt.figure(figsize=(16,4))
ax = plt.subplot(1,4,1)
ax.imshow(image)
ax.set_title('original image');

ax = plt.subplot(1,4,2)
ax.imshow(gray,cmap=cm.binary_r)
ax.set_axis_off()
ax.set_title('grayscale image');

ax = plt.subplot(1,4,3)
ax.imshow(cropped,cmap=cm.binary_r)
ax.set_axis_off()
ax.set_title('cropped image');

ax = plt.subplot(1,4,4)
ax.imshow(blurred,cmap=cm.binary_r)
ax.set_axis_off()
ax.set_title('blurred image');
#plt.imshow(gray,cmap=cm.binary_r)
```

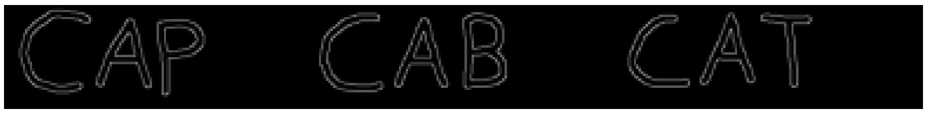
The output of this:



We right after this, perform an edge detection to show the model the edges of each letter. After that, it will sort the results from left to right.

```
# perform edge detection, find contours in the edge map, and sort the
# resulting contours from left-to-right
edged = cv2.Canny(blurred, 30, 250) #low_threshold, high_threshold
cnts = cv2.findContours(edged.copy(), cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_SIMPLE)
cnts = imutils.grab_contours(cnts)
cnts = sort_contours(cnts, method="left-to-right")[0]

figure = plt.figure(figsize=(7,7))
plt.axis('off');
plt.imshow(edged,cmap=cm.binary_r);
```

This is going to help our model to see where a letter ends and then it will try to figure out which letter is it according to the dataset.



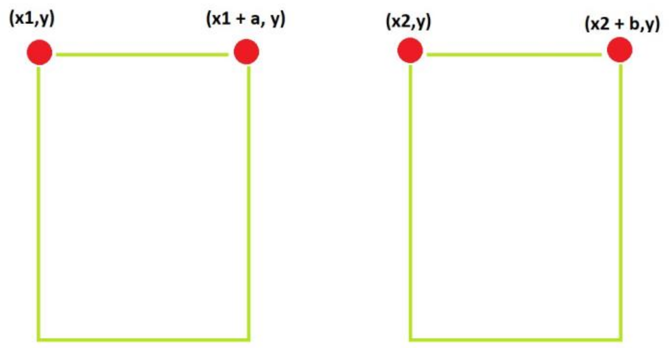
Now, before going any further we need to establish the way how we solved the issue of detecting the words of the given letters.

In our model as we've mentioned there is no "English Word" dataset that helped us with the detection of the words. We needed to come up with our own method to see what we can do to solve this issue. There were two methods we tried to solve this problem and we are going to try to explain them one by one:

METHOD 1 FOR WORD DETECTION:

The first method suggests that after we get the label boxes from the OpenCV library, we can measure the edge corner points. This way we can measure the difference between the boxes from the top edge points without complicating the results and also this way, we can create a code that is easy to read as well as easy to understand especially compared to the second model which we will talk about. The important part is to visualize how this method works so we can understand it better:

So, as we can see we have four points that we are working with. These points are (x1,y), (x1+a,y), (x2,y) and (x2+b,y) and let's talk about what these points mean.



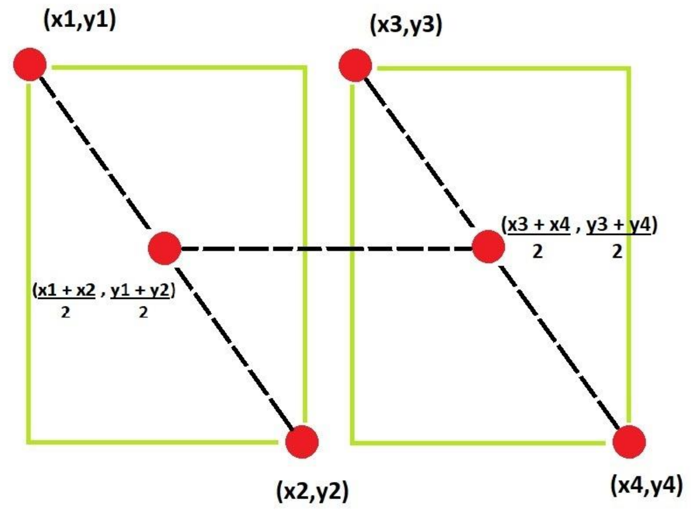
So, we said that we are going to calculate the distances between them using the corner points. According to that information we can say that if we want to find the end point of the first label box we can name it as (x1+a,y) since x1 is the top left corner and if the edge length between the left and the right top corner is a then this means that (x1+a) shows us the top left corner which is the endpoint of the first label box. With the same principle, we can say that x2 is the initial point or top right corner of the second letter's box. If we measure the distance between the **x1+a** and **x2** we can find the gap between these two letters. Additional to that, if we select a distance that will be more than the average than that means that if the gap between these two points is more than the average, we can end a certain word there and add an empty space between them.

So, the process of this model is like this:

We add each letter to a list to then mash-up. But before doing that, we need to find the gaps and in order to do that we are measuring the top right and left or ending point of the first letter box and initial point of the second letter box. In this scenario, we are not considering the margin of change of the y axes since we are not using it as a parameter for our model. So after finding the gaps we are adding the letters to a list and then add the necessary gaps or empty string gaps as elements in front of them. After doing this, we add the letters one by one to a string and get the result as a word with the intended gaps.

METHOD 2 FOR WORD DETECTION:

The second model suggests that if we find the center points of each letter box, we can measure the distances between these two points and if we detect an abnormal distance between these two points then we can divide the letter from there and mash the results afterwards.



Here we see the points. The x1,x2 and y1, y2 are for the first letter's box. We are going to find the center points as x1+x2 and take the average. We do the same for the y axis and we get the center point of the first letter's label box. For the second letter's label box, we do the same to get the center of the next letter's box. After that, we measure the distance between them and decide on whether we should put a gap between the letters or not. As we are gong to see, even though this is a method that is working properly without many issues for the instances that we create, there is a slight issue. It's adding an unnecessary complexity to our model. The way we want to do it is to do as easy to understand as possible, but this model is adding a new axis to the equation and also an unneeded calculation to the table. We don't want something like this since there is a better working way for our model. We are going to elaborate on our point with the next section of the model.

We tried to comment the code as much as we can but of course to clear up any misconceptions or vague topics, we are going to talk about the operations of the codes briefly.

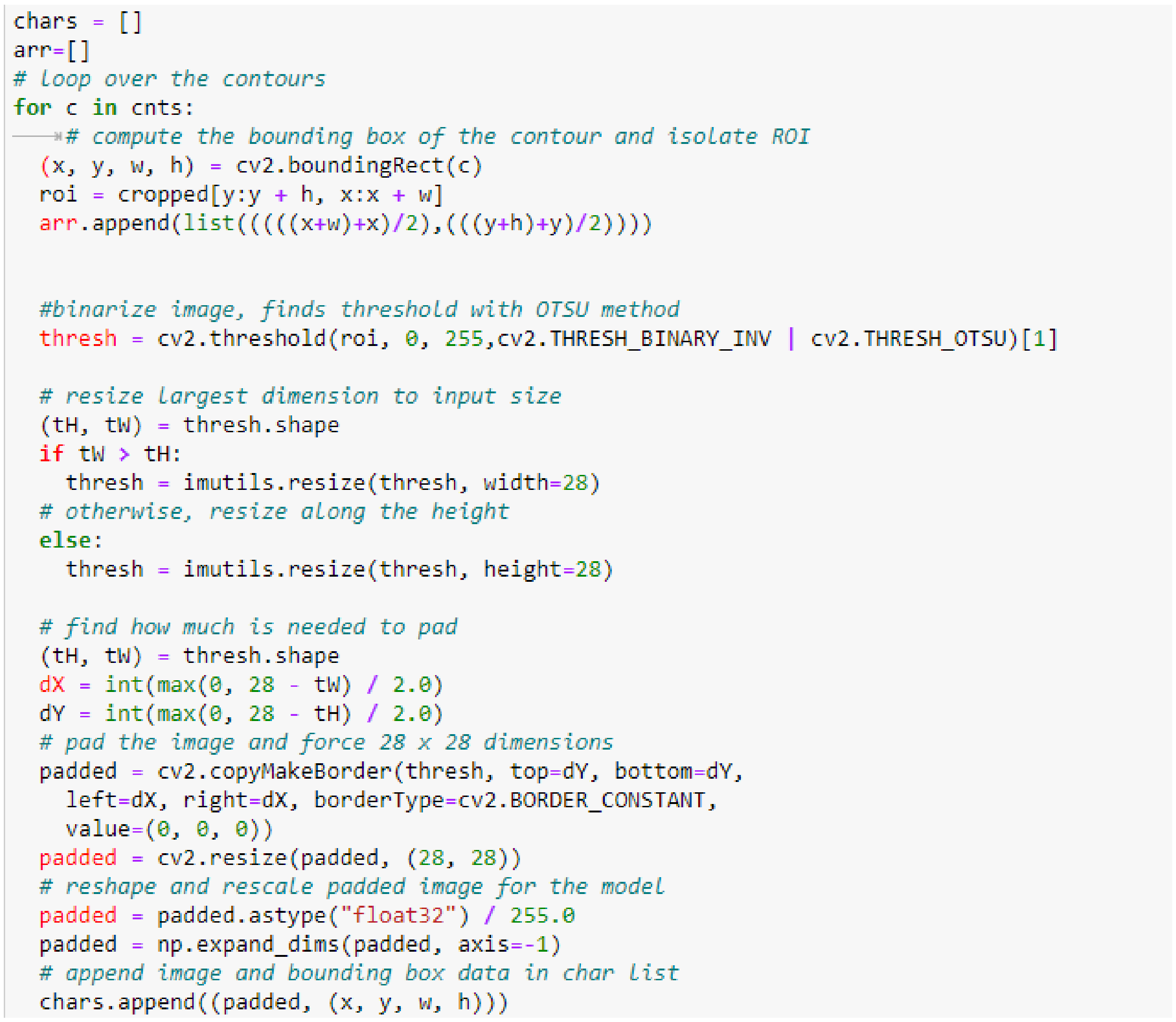
The performance of the models is going to be examined from the experiments part of this report with the images and graphs which will give us the insights of the model's performing measures.

Within the code snippet down below, we can see the method applied for the second model. Here we try to find the Region of Interests and then after we try to calculate the center points of the label boxes, we and append it to our list.

After that, we resize the images to suitable sizes and determine how much padding is needed. We pad the image for 28 by 28 dimensions ad reshape as well as rescale the padded images for the model that we are using.

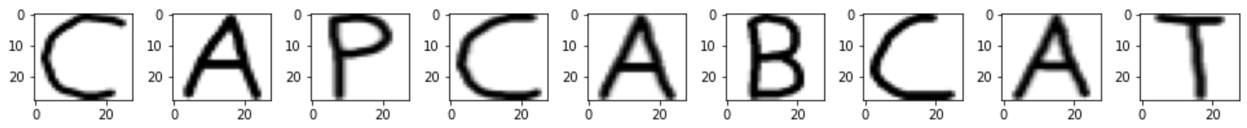
ONUR TOSUN 63180072 EMRE ALPOGUNC 63190011

Here is the code where arr is the list where we keep the distances between the center points.

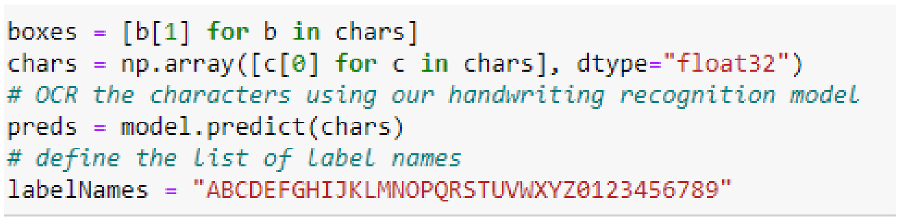


```
chars = []
arr=[]
# loop over the contours
for c in cnts:
  # compute the bounding box of the contour and isolate ROI
  (x, y, w, h) = cv2.boundingRect(c)
  roi = cropped[y:y + h, x:x + w]
  arr.append(list(((((x+w)+x)/2),(((y+h)+y)/2))))


  #binarize image, finds threshold with OTSU method
  thresh = cv2.threshold(roi, 0, 255,cv2.THRESH_BINARY_INV | cv2.THRESH_OTSU)[1]

  # resize largest dimension to input size
  (tH, tW) = thresh.shape
  if tW > tH:
    thresh = imutils.resize(thresh, width=28)
  # otherwise, resize along the height
  else:
    thresh = imutils.resize(thresh, height=28)

  # find how much is needed to pad
  (tH, tW) = thresh.shape
  dX = int(max(0, 28 - tW) / 2.0)
  dY = int(max(0, 28 - tH) / 2.0)
  # pad the image and force 28 x 28 dimensions
  padded = cv2.copyMakeBorder(thresh, top=dY, bottom=dY,
    left=dX, right=dX, borderType=cv2.BORDER_CONSTANT,
    value=(0, 0, 0))
  padded = cv2.resize(padded, (28, 28))
  # reshape and rescale padded image for the model
  padded = padded.astype("float32") / 255.0
  padded = np.expand_dims(padded, axis=-1)
  # append image and bounding box data in char list
  chars.append((padded, (x, y, w, h)))
```

Then we plot the characters that we detect from this model:

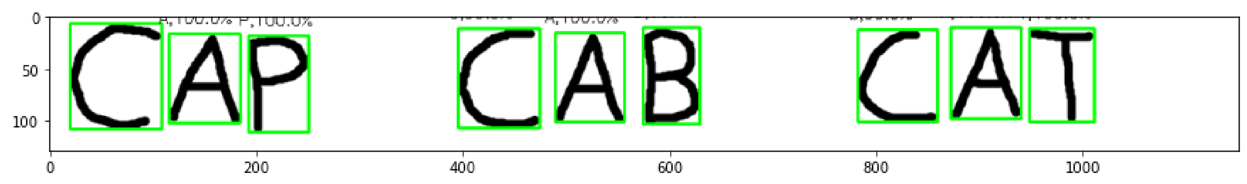


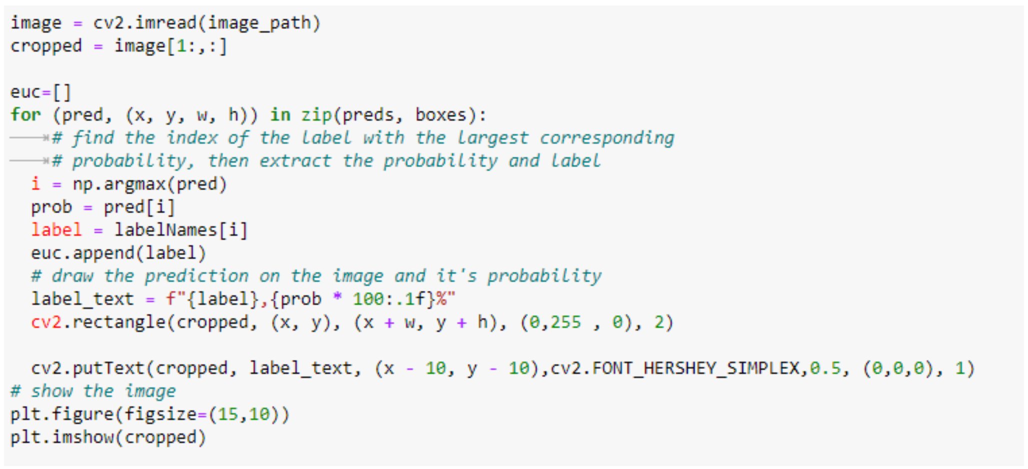
Create the label names as well as the prediction boxes:



```
boxes = [b[1] for b in chars]
chars = np.array([c[0] for c in chars], dtype="float32")
# OCR the characters using our handwriting recognition model
preds = model.predict(chars)
# define the list of label names
labelNames = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
```

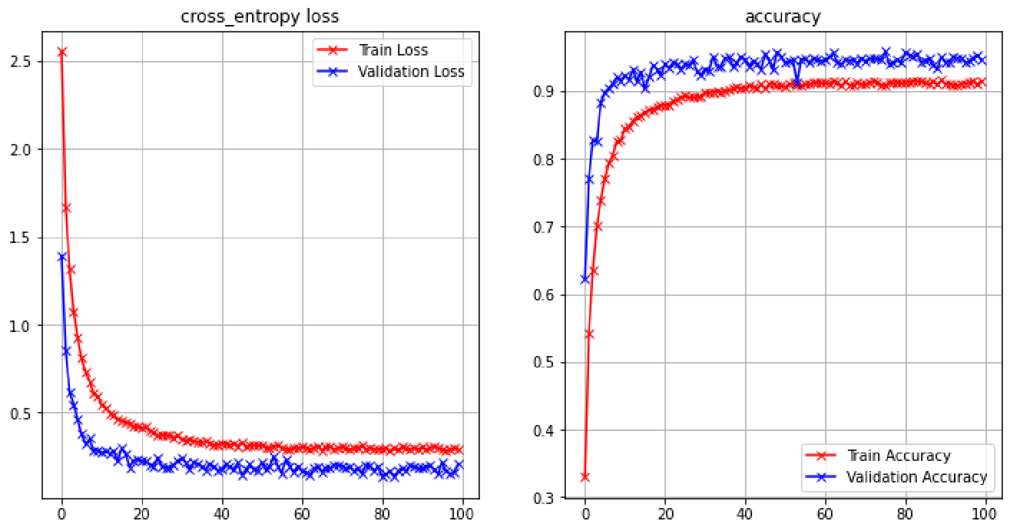
ONUR TOSUN 63180072 EMRE ALPOGUNC 63190011





```
image = cv2.imread(image_path)
cropped = image[1:,:]

euc=[]
for (pred, (x, y, w, h)) in zip(preds, boxes):
    # find the index of the label with the largest corresponding
    # probability, then extract the probability and label
  i = np.argmax(pred)
  prob = pred[i]
  label = labelNames[i]
  euc.append(label)
  # draw the prediction on the image and it's probability
  label_text = f"{label},{prob * 100:.1f}%"
  cv2.rectangle(cropped, (x, y), (x + w, y + h), (0,255 , 0), 2)

  cv2.putText(cropped, label_text, (x - 10, y - 10),cv2.FONT_HERSHEY_SIMPLEX,0.5, (0,0,0), 1)
# show the image
plt.figure(figsize=(15,10))
plt.imshow(cropped)
```

The output with the code that which it does the operation can be found here. This part of the code is for the finding the indexes with the biggest probability from the dataset and get the result and label the letters accordingly. This is does the part where we finish the letter detection and after that we print the loss and the accuracy of this model:



This is the performance measures of the second model with the given sample image.

Here, we have the code snippet where we do the operation of the second method. The output of this snippet can also be found as well:

```
print(arr)

foo=[]

d= np.diff(arr,axis=0)
segdists = np.hypot(d[:,0],d[:,1])
segdists=np.sqrt((d**2).sum(axis=1))
print(segdists)

print(len(segdists))
segUpd=list(segdists)
segUpd.append(0)
for i,j in zip(range(len(segUpd)),range(len(euc))):
  foo.append(euc[j])
  if segUpd[i]>120:
    foo.append(" ")
output=""
for i in foo:
    output+=i



print(output)
```

```
[[64.5, 57.0], [150.5, 59.0], [222.0, 64.5], [435.5, 59.0], [523.5, 58.0], [602.5, 56.5], [821.5, 56.5], [907.0, 54.0], [980.
56.0]]
[ 86.02325267  71.71122646 213.57083134  88.00568163  79.01423922
 219.          85.5365419   73.52720585]
8
CAP CAB CAT
```

Here, we create an empty list to add the letters as well the gaps where we find them suitable. We measure the distances between the 2 consecutive boxes and add go the next two boxes afterwards. When we print **segdists** we can see that clearly, the first lines of the code are for the corner points where we get it by printing **the arr** list. After that we print the distances between them (each consecutive 2 boxes) and after that we say that if the gaps that we measure is bigger than 120 pixels then we are going to out a gap between them and add the results to a **outputs** string which is going to be our final result.

In the end, we can see the answer clearly which means that the model is working properly.

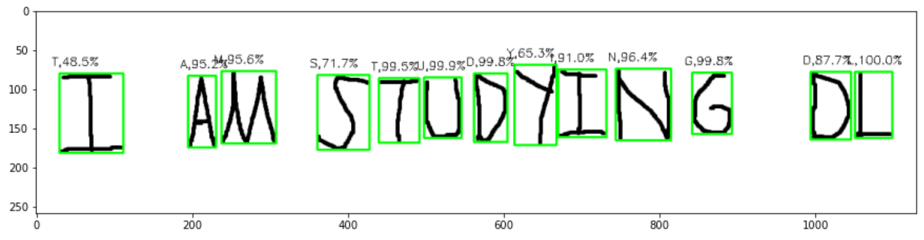
Last but not least, we can show the translated version of the output string using a library called GoogleTranslator from deep_trasnslator:

```
from deep_translator import GoogleTranslator
print(GoogleTranslator('auto','tr').translate(output))
```

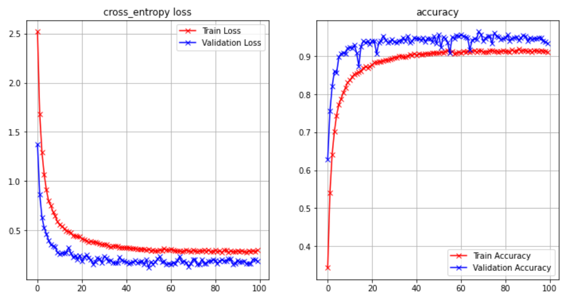
```
KAP KABİN KEDİ
```

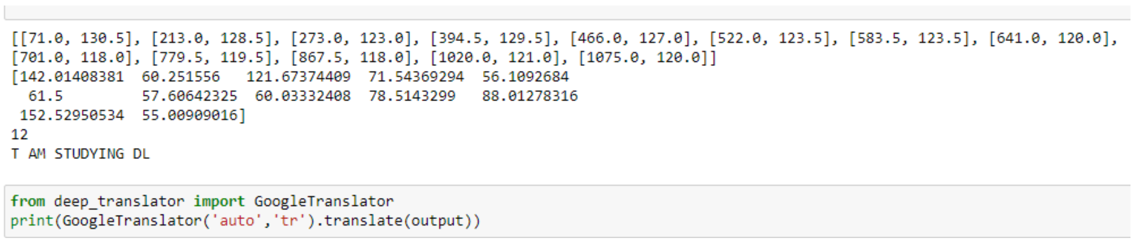
EXPERIMENTS:

Here are the results we get from the second model with Epoch=100,Batch Size=32,Optimizer=RMS Prop:



ONUR TOSUN 63180072 EMRE ALPOGUNC 63190011

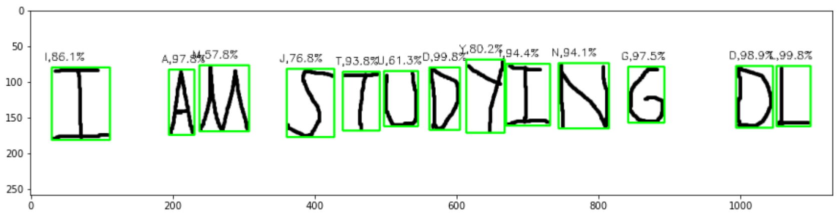


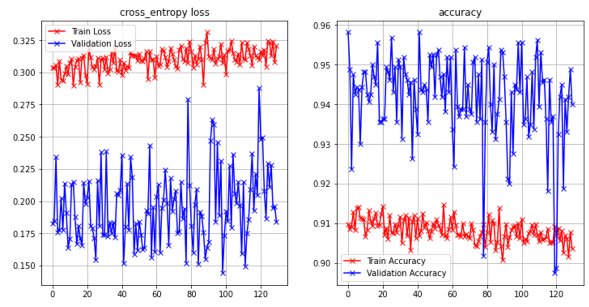


```
[[71.0, 130.5], [213.0, 128.5], [273.0, 123.0], [394.5, 129.5], [466.0, 127.0], [522.0, 123.5], [583.5, 123.5], [641.0, 120.0],
[701.0, 118.0], [779.5, 119.5], [867.5, 118.0], [1020.0, 121.0], [1075.0, 120.0]]
[142.01408381  60.251556   121.67374409  71.54369294  56.1092684
  61.5         57.60642325  60.03332408  78.5143299   88.01278316
 152.52950534  55.00909016]
12
T AM STUDYING DL
```

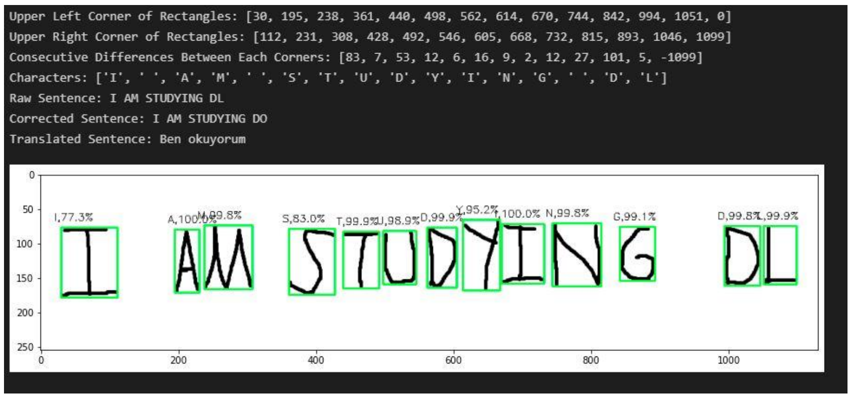
```
7]: from deep_translator import GoogleTranslator
    print(GoogleTranslator('auto','tr').translate(output))
```

Second model with Epoch=130,Batch Size=32,Optimizer=RMS Prop:



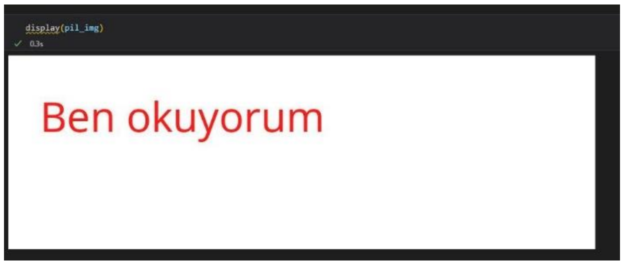


First model with Epoch=100,Batch Size=32,Optimizer=RMS Prop:



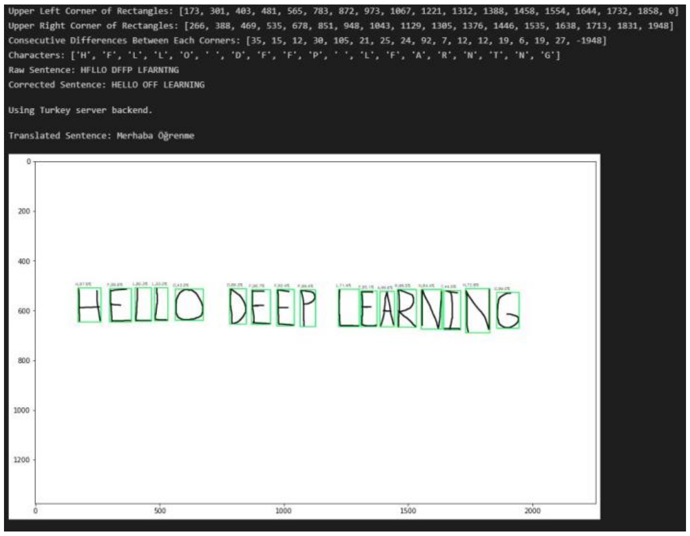
```
Upper Left Corner of Rectangles: [30, 195, 238, 361, 440, 498, 562, 614, 670, 744, 842, 994, 1051, 0]
Upper Right Corner of Rectangles: [112, 231, 308, 428, 492, 546, 605, 668, 732, 815, 893, 1046, 1099]
Consecutive Differences Between Each Corners: [83, 7, 53, 12, 6, 16, 9, 2, 12, 27, 101, 5, -1099]
Characters: ['I', ' ', 'A', 'M', ' ', 'S', 'T', 'U', 'D', 'Y', 'I', 'N', 'G', ' ', 'D', 'L']
Raw Sentence: I AM STUDYING DL
Corrected Sentence: I AM STUDYING DO
Translated Sentence: Ben okuyorum
```

ONUR TOSUN 63180072 EMRE ALPOGUNC 63190011

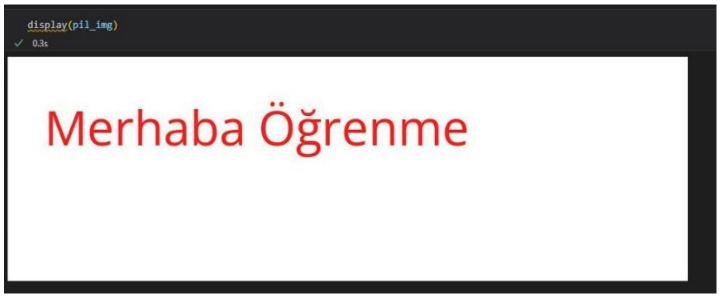


```
display(pil_img)
✓ 0.3s
```

Ben okuyorum



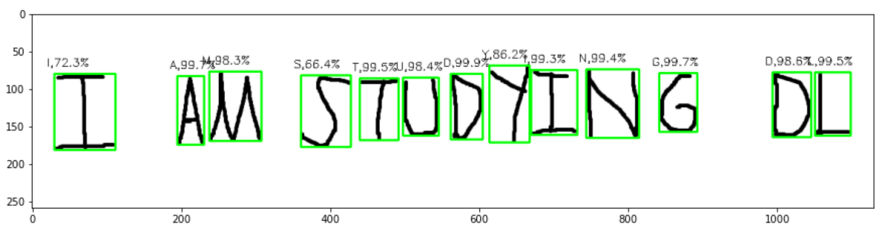
```
Upper Left Corner of Rectangles: [173, 301, 403, 481, 565, 783, 872, 973, 1067, 1221, 1312, 1388, 1458, 1554, 1644, 1732, 1858, 0]
Upper Right Corner of Rectangles: [266, 388, 469, 535, 678, 851, 948, 1043, 1129, 1305, 1376, 1446, 1535, 1638, 1713, 1831, 1948]
Consecutive Differences Between Each Corners: [35, 15, 12, 30, 105, 21, 25, 24, 92, 7, 12, 12, 19, 6, 19, 27, -1948]
Characters: ['H', 'F', 'L', 'L', 'O', ' ', 'D', 'F', 'F', 'P', ' ', 'L', 'F', 'A', 'R', 'N', 'T', 'N', 'G']
Raw Sentence: HFLLO DFFP LFARNTNG
Corrected Sentence: HELLO OFF LEARNING

Using Turkey server backend.

Translated Sentence: Merhaba Öğrenme
```



```
display(pil_img)
✓ 0.3s
```

Merhaba Öğrenme

Second model with Epoch=100,Batch Size=32,Optimizer=ADAM:



[[71.0, 130.5], [213.0, 128.5], [273.0, 123.0], [394.5, 129.5], [466.0, 127.0], [522.0, 123.5], [583.5, 123.5], [641.0, 120.0], [701.0, 118.0], [779.5, 119.5], [867.5, 118.0], [1020.0, 121.0], [1075.0, 120.0]] [142.01408381 60.251556 121.67374409 71.54369294 56.1092684 61.5 57.60642325 60.03332408 78.5143299 88.01278316 152.52950534 55.00909016]
12
I AMSTUDYING DL

DL okuyorum

So, here we can see the results and we want to talk about them:

In the results first we tried to make the second model better since we were trying to make the second model so good that we should be able to choose it compared to the first model despite the unnecessary complexity. We tried different hyper-parameters and, in the end, these were the most remarkable results that we could get. We can see that model 2 is performing quite good but when we try to run it more than 100 epochs the model starts to corrupt and unfortunately it becomes unusable. Then we tried different optimizers and Adam optimizers really did a god job but then, we compared the results from one to anther and we saw that the results were nearly identical. Then we asked this question, if they are both performing identical why use the model with more complexity and more difficult to deal with?

This question was especially crucial since within the last part we mention that we want to make this code perfect by adding more attributes such as multiple line reading. Thus, we made the choice that we should use the first model.

These are not all of the experiments that we made. We've have added some of the additional ways that we couldn't mention here to the zip file as well.

To talk about the negative parts of these models is that there is no way to overcome some faulty recognitions such as recognizing I as 1 or M as V. We tried various ways to overcome this but unfortunately, we could overcome this problem for some measure but it's not 100% gone.

The other problem is that we needed to determine the pixel gaps between them manually. After making some trial and errors, we concluded that 35 pixels was the maximum distance that can be accepted as a gap within a word. If the gap is more than that, we classify it as a different word. But even though this 35-pixel gap works quite good for the first model, it's not working that good for the second model. We determined that around 100 pixels were good for the second model but again this proves our point that why we should use the first model rather than the second model. The first model has easier way to deal with this problem as well.

CONCLUSION:

To sum things up, we have explained our problem that we want to deal with the ways how we wanted to solve this problem. Even though both of the methods work somewhat accurate there are of course some problems with our model like not being able to do the scanning of words that are multiple lined. But in the end, we believe that our model performed good enough to prove our point where we can apply the word detection that we are dealing with without use of any additional libraries. We can try to add additional methods to improve our model but our model we think that works consistent enough. The difficult part was to implement the ways that we mentioned as word detection sections that were once theoretical only. How can we implement such ways to our code was the main question, but we believe that we dealt with it? The next thing we would like to implement would be to add additionally line support. We want to go further with this project and later on, we plan to share the "perfected" version on GitHub.

REFERENCES

1. Javidi, Bahram. Image recognition and classification: algorithms, systems, and applications. CRC press, 2002.
2. Yan, L. C., Yoshua, B., & Geoffrey, H. (2015). Deep learning. nature, 521(7553), 436-444.
3. Shen, Dinggang, Guorong Wu, and Heung-Il Suk. "Deep learning in medical image analysis." Annual review of biomedical engineering 19 (2017): 221-248.
4. Gal, Yarin, Riashat Islam, and Zoubin Ghahramani. "Deep bayesian active learning with image data." In International Conference on Machine Learning, pp. 1183-1192. PMLR, 2017.
5. Wu, Meiyin, and Li Chen. "Image recognition based on deep learning." In 2015 Chinese Automation Congress (CAC), pp. 542-546. IEEE, 2015.
6. Pak, Myeongsuk, and Sanghoon Kim. "A review of deep learning in image recognition." In 2017 4th international conference on computer applications and information processing technology (CAIPT), pp. 1-3. IEEE, 2017.
7. Wang, H., Z. Lei, X. Zhang, B. Zhou, and J. Peng. "Machine learning basics." Deep learning (2016): 98-164.
8. O'Shea, Keiron, and Ryan Nash. "An introduction to convolutional neural networks." arXiv preprint arXiv:1511.08458 (2015).
9. Albawi, Saad, Tareq Abed Mohammed, and Saad Al-Zawi. "Understanding of a convolutional neural network." In 2017 international conference on engineering and technology (ICET), pp. 1-6. Ieee, 2017.
10. Rawat, Waseem, and Zenghui Wang. "Deep convolutional neural networks for image classification: A comprehensive review." Neural computation 29, no. 9 (2017): 2352-2449.
11. Chowdhary, KR1442. "Natural language processing." Fundamentals of artificial intelligence (2020): 603-649.
12. Allen, James F. "Natural language processing." In Encyclopedia of computer science, pp. 1218-1222. 2003.
13. Mustafa, Mohamed Elhafiz, and Murtada Khalafallah Elbashir. "A deep learning approach for handwritten Arabic names recognition." Int. J. Adv. Comput. Sci. Appl 11, no. 1 (2020): 678-682.
14. Dewa, Chandra Kusuma, Amanda Lailatul Fadhilah, and A. Afiahayati. "Convolutional neural networks for handwritten Javanese character recognition." IJCCS (Indonesian Journal of Computing and Cybernetics Systems) 12, no. 1 (2018): 83-94.