**Handwriting detection**

**The previous sprint**

We trained a handwriting recognition model based on: MNIST 0-9, Kaggle A-Z using: Keras, Tensorflow and Deep Learning.

**Current work**

We used the previous model to recognize our own images.

The project so far consists of:

A module for I / O helper functions and for the recognition model (implementation of the ResNet learning model), a CSV file that contains the Kaggle A-Z dataset, the recognition-driven model, a graph in which we display the latest results and the main application in which we import everything and feed the image to the model.

We also use a package called immutlis that makes functions such as translation, rotation, resizing, skeletalization more convenient as well as easier display of Matplotlib images with OpenCV and Python.

After importing everything we need, we convert the image to grayscale:

gray = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

and then apply a Gaussian blur to reduce noise:

blurred = cv2.GaussianBlur(gray, (5, 5), 0)

From here, we detect the edges of the blurred image and then locate the outline of the characters and initialize the list that will hold them.

cnts = cv2.findContours(edged.copy(), cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)

After that each contour goes through 4 steps:

1. We select the contours of reasonable dimensions and we extract the region of interest (roi). To do that we take the bounding box information

(x, y, w, h) = cv2.boundingRect(c)

and we put new delimitations on how big or small the contours we want them to be and take the ones that fit the parameters.

roi = gray[y:y + h, x:x + w]

1. Clean the image using a threshold algorithm (Otsu) to have an image with white characters on a black background

thresh = cv2.threshold(roi, 0, 255,

cv2.THRESH\_BINARY\_INV | cv2.THRESH\_OTSU)[1]

(tH, tW) = thresh.shape

1. Resize the characters to a 32x32 pixel image and then centralize the outline

if tW > tH:

thresh = imutils.resize(thresh, width=32)

else:

thresh = imutils.resize(thresh, height=32)

1. Scale the pixel intensity between [0,1] and set a size for the batch then wrap the cleaned outline together with the bounding box in a double tuple and add it to the character list.

padded = padded.astype("float32") / 255.0

padded = np.expand\_dims(padded, axis=-1)

After going through all the characters and finishing the list, we feed it into the model to make predictions, then we will look at each prediction and its bounding box in a loop and take the highest probability.

for (pred, (x, y, w, h)) in zip(preds, boxes):

i = np.argmax(pred)

prob = pred[i]

**Preliminary performance measurements**

**Text, letter, whiteboard

Description automatically generated**

**Text, letter, whiteboard

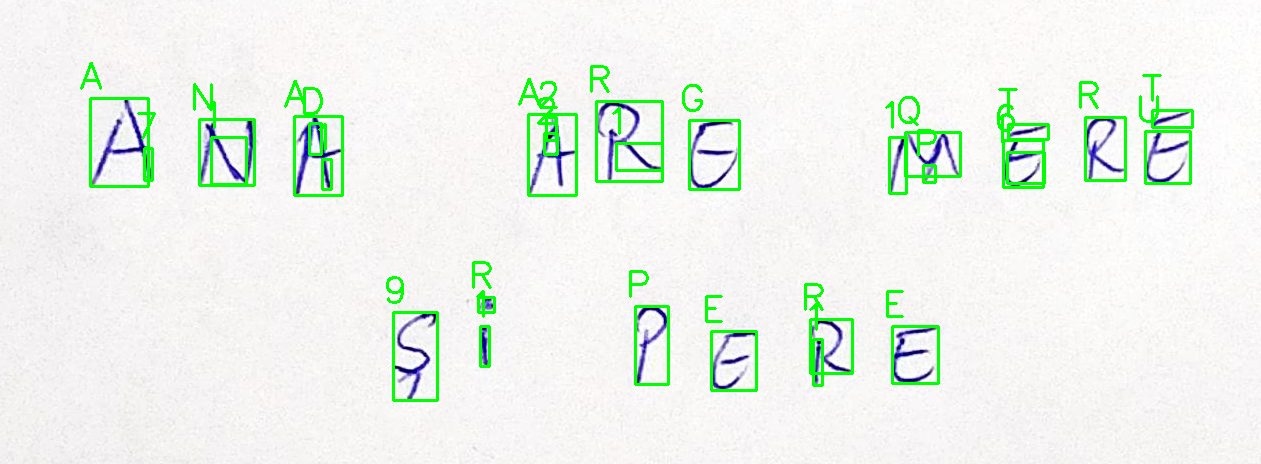
Description automatically generated**

For the picture with "Hello world" our recognition model worked well, but made two mistakes.

In the second image we have only one wrong digit (1 confused with 7).

While our handwriting recognition model achieved 96-97% accuracy on our test set, our handwriting recognition accuracy on our own custom images is slightly lower than this.

For example, in the following image we have the worst result, even if the model has a good prediction on a character, due to the way it is written, sometimes it sees a single letter as several and the result is unrecognizable.

****

**Conclusions and future work**

Given the results, the problem is much more complicated than we expected, especially since our recognition model did not even take into account the case where the characters can be connected, which means that several joined characters will be treated as one, and this would result in wrong predictions from the model created.

The treatment of connected characters is another open area of research in the field of computer vision; however, there are models that look promising to improve the accuracy of handwriting recognition such as: LSTM (Long Short-Term Memory).