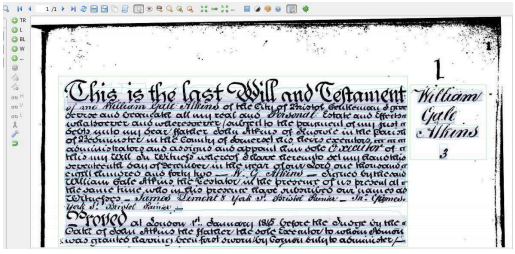
Handwriting Text Recognition (HTR)

Link to Repository: https://github.coecis.cornell.edu/nm458/4701-project

Introduction:

Meet Em, a diligent college student and an SDS note taker! Her job is to take notes in various classes so they can be distributed to students with SDS accommodations who need access to learning materials. The notes that will ultimately be sent to the students must be digital, but Em prefers taking handwritten notes with a pencil and notebook. In order to solve Em's problem, we hope to develop an AI Handwriting to Text model that allows Em to take notes to the best of her ability while ensuring the notes are accessible for the students who need them.

We hope to use our model to help mitigate accessibility issues related to handwritten documents. Making handwritten notes accessible to a larger pool of users by translating them into typed forms of communication can be powerful in expanding accessibility. The clarity and visibility of a document with this effect immediately increases for users, especially for those with disabilities who may have been unable to properly view and understand the original document (i.e. due to colors, handwriting style, size). We have described one example of such a problem above, but we also believe that the scope can be broadly applied to other disciplines, such as research.



For example, an application we believe is important is transcribing ancient texts and related documents, some of which may be unreadable to the human eye. However, by implementing a well trained AI model, such texts can be transcribed, preserved and studied by anthropologists, philosophers, and historians. Using this technology to decipher and reconstruct ancient texts into documents for greater visibility can allow research on these texts to become accessible for translations and other forms of analysis. AI Transcription may also increase the amount of translations available for research by improving the access of these documents.We believe that it is important to maintain as much documentation as possible from history and hope to use the Handwriting to Text model to do so.

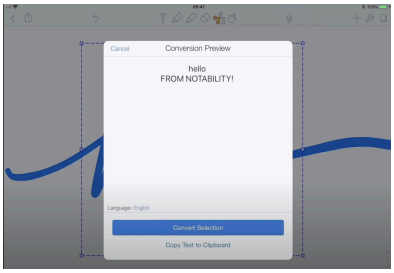
Our vision for creating such a program is to experiment with these potential outcomes and better understand how such technology can be applied to Cornell and beyond. We hope to modernize the way

people use handwritten notes and provide a tool that can make notes universally accessible and utilizable. We also hope to learn about how each level of knowledge we achieve in creating this model could help us increase documentation accessibility.

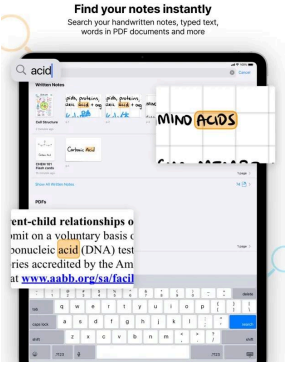
Prior Work:

Of prior work, two of the most notable existing apps are Notability and GoodNotes 5. Notability allows users to take handwritten notes on touchscreen devices, and allows users to search for keywords within their handwritten notes. Notability advertises that their Handwriting Text Recongition and Search are powered by MySript (Notability). While the exact implementation used by Notability is not open source, it is likely that they use similar AI methods to develop and train a model to recognize handwritten letters and numbers and detect words.

Examples of Handwriting Tech Recognition in Notability (YouTube Demo)



GoodNotes 5 is similar to Notability in that it allows users to write and share written documents comfortably as they take notes on tablets or other touchscreen devices, but it differs in variety of features. Among the two apps, GoodNotes 5 allows users much more flexibility in the shape, size, color, and texture of the pen used, whereas Notability is far more restrictive in this aspect. It is important to consider the variety of features, which can indicate a model that has been trained with more diverse data, allowing users to be more creative while preventing loss of detection of letters, numbers, and words.



In terms of existing technology, we researched heavily into past models to determine which model we should implement. Among the most notable are Keras and TensorFlow, a model by Harald Scheild, and Arthurflor. Keras is a deep learning Python API which runs on top of TensorFlow, a machine learning platform. Keras enables fast experimentation, which we believed to be important to ensure we would be able to complete this project in the given time frame. Keras has been used to develop handwriting

recognition models using the IAM Dataset, which contains multiple forms of handwritten English text which can be used to train handwriting recognition models. Since the IAM Dataset is used widely, we believed the Keras implementation could serve as a good starting point to learn how to build our model.

The handwriting recognition model built by Harald Scheild uses TensorFlow, and the implementation is minimalistic, uses neural networks, and can be trained on the CPU. Handwritten recognition models scan images to transcribe handwritten into digital text. This can be achieved using a neural network that is trained on images from the IAM dataset. By keeping the input layer for the neural network small, training this model is feasible on the CPU. This implementation uses TensorFlow to achieve this goal. We believed this model was feasible because it does not require a GPU, and is within the scope of our abilities, time frame, and hardware to train.

Another implementation we looked into was Arthurflor, which is a Handwritten Text Recognition (HRT) system that is implemented using TensorFlow. This system is also a Neural Network model (similar to above) that is trained on the IAM dataset. However, this implementation is also trained on various other offline handwriting text recognition datasets, such as Bentham, Rimes, Saint Gall, and Washington. The model, unlike the one above, recognizes the text of segmented text lines in images. The model also used data partitioning to train, validate, and test each dataset.

Methods:

Our approach was to use deep learning (specifically a convolutional neural network) to build our model. The core vision was to develop an Upload and Translate model, in which images of handwritten text could be uploaded, and our trained model would attempt to scan the image and then translate the handwritten text.

For our first model, we built on a Handwritten Digits Recognition YouTube tutorial that uses the MNIST dataset from the Keras library in Python to first be able to recognize and translate digits. We wanted to focus on this approach first as thereare only ten digits as opposed to twenty six letters, and we believed it was important to be able to flesh out our understanding on a deep learning model with a smaller dataset before moving onto a larger dataset. Of the seventy thousand images in the MNIST dataset, we used sixty thousand images for training and ten thousand images for validating. We wanted to ensure there would be enough samples in the training dataset and validation dataset so our model could be properly fitted.

The model we used for this portion was Sequential, as it allowed us to build a model layer by layer compared to a function API. We used three convolution layers to deal with our input images as well as an activation function. The first layer takes in the input image after which our model calls the activation function (RELU to ensure non-linearity). There are two additional layers both of which also call the activation function immediately after (RELU for non-linearity or softmax for class probabilities). After we finish convolving, we use a Flatten layer which serves as a connection in between the convolution and dense layers. Dense layers are a standard neural network layer type, which is what we used for output layer.

As seen in the diagram below, we have eighty one thousand and sixty six trainable parameters. The diagram visualizes our sequential model, which uses convolutional layers as well as dense neural network layers.

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_1 (Conv2D)           (None, 26, 26, 64)        640

 activation_1 (Activation)   (None, 26, 26, 64)        0

 max_pooling2d_1 (MaxPooling  (None, 13, 13, 64)       0
 2D)

 conv2d_2 (Conv2D)           (None, 11, 11, 64)        36928

 activation_2 (Activation)   (None, 11, 11, 64)        0

 max_pooling2d_2 (MaxPooling  (None, 5, 5, 64)         0
 2D)

 conv2d_3 (Conv2D)           (None, 3, 3, 64)          36928

 activation_3 (Activation)   (None, 3, 3, 64)          0

 max_pooling2d_3 (MaxPooling  (None, 1, 1, 64)         0
 2D)

 flatten (Flatten)           (None, 64)                0
```

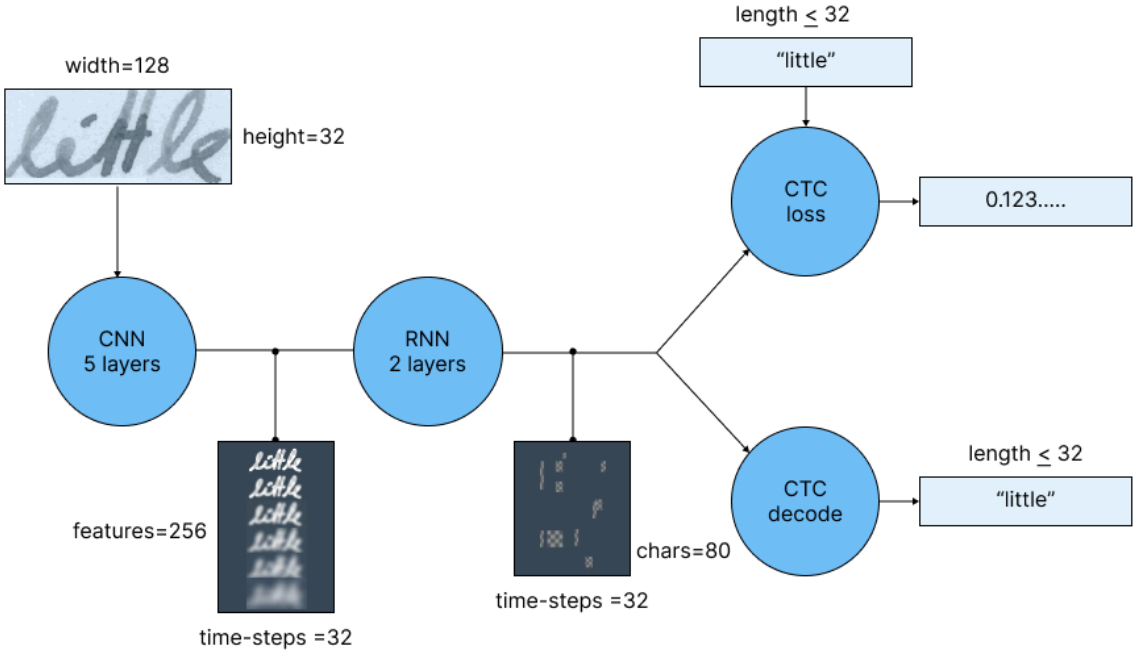
```
 dense (Dense)               (None, 64)                4160

 activation_4 (Activation)   (None, 64)                0

 dense_1 (Dense)             (None, 32)                2080

 activation_5 (Activation)   (None, 32)                0

 dense_2 (Dense)             (None, 10)                330

 activation_6 (Activation)   (None, 10)                0

=================================================================
Total params: 81,066
Trainable params: 81,066
Non-trainable params: 0
_________________________________________________________________
```

We then compile our model using optimizer, loss, and metrics parameters. The optimizer parameter controls the learning rate, which determines how fast the optimal weights for the model are calculated (smaller learning rate may lead to more accurate weights but computation time for weights will be longer). We used ‘adam’ for our optimizer which adjusts the learning rate as the model trains. For our loss function, we used ‘sparse_categorical_crossentropy’ for which a lower score indicate a better performing model. We also used the accuracy metric to see the accuracy score on our validation set when we train and validate our model. These are also the parameters with which we evaluated our approach, as we believed it was important to take into account all three metrics to gain a better understanding of the efficiency of our system. The quantitative measures we focused on the most were the loss value and the accuracy score (we wanted to minimize loss and maximize accuracy).

To train our model, we use the ‘fit()’ function which takes in our training data, target data, number of epochs, and validation split. We used five epochs, as we believe cycling through the data five times will allow the model to improve. We believed using a higher number may deteriorate our model and disallow it from improving.

Finally, we can use our test data and have our model make predictions. We use a probability array to make predictions on the likelihood that an input image represents one of ten digits, and the digit corresponding to highest probability value will be the predicted number.

Our second model is also a neural network model, but this model consisted of convolution neural network (CNN) layers, recurrent neural network (RNN) layers, as well as a Connectionist Temporal Classification (CTC) layer. We built on the tutorial made by Harald Scheidl. An overview of the system is shown below.



The model is trained on images from the IAM dataset, which allows us to have a smaller input layer, making neural network training feasible on a CPU using TensorFlow.

First, we feed an input image into the CNN layers, which are trained by our model to detect important features. Each layer has a convolution operation, a non-linear Rectified Linear Unit function (RELU), and a pooling layer. The convolution operation applies a filter kernel of size 5x5 for the first two layers and 3x3 for the last three layers. Similar to our first model for digits, we the RELU activation function to ensure non-linearity. We lastly use a pooling layer to summarize the regions of the image as well as output a downsized version of our input image (channels are however added throughout this process).

Next, we use RNN to propagate important information through a specific feature sequence (ours contains two hundred fifty six features for every time step). We used the Long Short-Term Memory (LSTM) implementation as it allows us to train our model in a more robust manner.

Finally, we use CTC. While we train the neural network, we use the RNN output (character probability matrix) as well as the ground truth text for CTC inputs to compute the loss value. We use the mean of the loss values of an inferred batch to train the neural network.

Our model consisted of four modules. The first module was our preprocessing module, which dealt with image processing. The module prepared images from the IAM dataset so they could be used as inputs for the neural network. The second module our data loading module, with dealt with file I/O. This module read samples and placed them into batches. It also provided an iterator interface which could be used to traverse the data. The third module created our actual model (implementation described above). This module loads and saves modules and manages the TensorFlow sessions. It also provided us with an interface for training and inferring. The last module was our main module, which combined all previous modules and provided us with an interface to train, validate, and test the model.

The main difference between our first two modules was the inclusion of the RNN layers as well as the CTC. The CTC improves accuracy by allowing us to train our model using the loss values from our training dataset. The quantitative metrics we used for our model were the character error rate (percentage of incorrectly recognized characters over total number of input characters) and word accuracy (percentage of recognized words over total number of input words).

The main focus of our evaluation was the generalizability of our models. We wanted to be able to write letters and numbers in our own handwriting styles on different devices and materials, and have our model be capable of recognizing the handwritten text. As we mentioned previously, one of our main goals for creating this model was to improve accessibility. In order to ensure we are able to work towards this goal, we need to be able to generalize our model to various inputs outside of those contained in the datasets we used to train our model.

In order to compare the two models, we focused on the accuracy metric as a quantitative measure. We believe this would be the best metric to evaluate the accuracy of our models, as we wish to maximize accuracy. This also aligned with our goal of generalizability, as a higher accuracy rate would ensure increased visibility and clarity on a sample handwritten text.

Results:

Digits Demo: After training and testing on the mnist dataset, we created our own digit bank from (0 to 9) to test our model. We handwrote the digits on black paper with white ink on notability to run our model.

We printed the COMPUTER'S BEST GUESS and the PROBABILITY for said guess since we had the probability that the computer corresponded each digit with a digit between [0,9]. We double checked accuracy with image display after the print statements. It correctly identified 14/16 inputs giving it an 87.5% accuracy on the custom digit bank we created for this project.

Accuracy: 0.9847 (after 5 epochs)

Computer's Best Guess: 0 ✅

Probability: 0.7876914

Computer's Best Guess: 0 ✅

Probability: 0.9997167

KEY

✅Correct
❌Incorrect

Computer's Best Guess: 1 ✅

Probability: 1.0

Computer's Best Guess: 1 ✅

Probability: 1.0

Computer's Best Guess: 2 ✅

Probability: 0.9999944

Computer's Best Guess: 2 ✅

Probability: 0.99996305

Computer's Best Guess: 3 ✅

Probability: 0.99999976

Computer's Best Guess: 3 ✅

Probability: 0.9955836

Computer's Best Guess: 4 ✅

Probability: 0.9952188

Computer's Best Guess: 1 ❌

Probability: 0.99992704

Computer's Best Guess: 5 ✅

Probability: 0.99987125

Computer's Best Guess: 6 ✅

Probability: 0.9999999

Computer's Best Guess: 1 ❌

Probability: 0.99999046

Computer's Best Guess: 7 ✅

Probability: 0.9915236

Computer's Best Guess: 8 ✅

Probability: 0.99994195

Computer's Best Guess: 9 ✅

Probability: 0.964976

Aside: Testing our model with different fonts of typed text.

During our research into prior art, we saw that models for handwriting text recognition were also being used to recognize typed text on photos. For example, Apple uses it in their app Preview to enhance user experience by connecting addresses to their Maps application and phone numbers to their Phone application.

There are four main types of fonts: Serif, Sans Serif, Script, Decorative. We tested our digits model on each of these to see the accuracy. Sans serif fonts are typically more used for display but blank had the highest

Sans Serif Font

Digit										
Result	✅	✅	✅	✅	❌	✅	✅	❌	✅	✅

Serif Font

Digit										
Result	✅	✅	✅	✅	❌	✅	✅	✅	✅	✅

Script Font (closer to handwriting)

Digit										
Result	❌	✅	❌	✅	❌	✅	❌	❌	✅	✅

With the aside and our custom dataset, we can see our model greatly struggles with identifying 7, oftentimes misidentifying it with 1.

Main Model Test Dataset:

The testing set for our character and word model yielded a character error rate of 10.624916% and a word accuracy rate of 73.686957%. The character error rate is the proportion of incorrectly recognized characters to the total number of characters in the test dataset. The character error rate implies a character accuracy rate of 89.375084%. The word accuracy rate is the proportion of correctly recognized words to the total number of words in the test dataset.

Main Model Sample Data:

Running our model on the sample PNG with the word “little” from the IAM dataset yielded a Recognized value of “little” (which is correct!) and a Probability value of 0.9662546, which is the probability that the Recognized value is accurate.

Recognized: "little"

Probability: 0.9662546

Main Model Custom Data:

We created custom input images by writing our names on paper, scanning them with our iPhone and converting the PDFs to PNGs. Below is a table of the four custom input images with Recognized (the output translation from our model) and Probability (the probability that the recognized word is accurate) values. The character error rate is 13.04% and the word accuracy rate is 50%

Recognized: "heysil" ✅

Probability: 0.36286518

Recognized: "keri" ✅

Probability: 0.49379802

Recognized: "meta" ❌

Probability: 0.3235437

Recognized: "dluwatise" ❌

Probability: 0.38318378

Discussion:

We used test driven development to build our system. We essentially specified test cases and built the different components of our model iteratively. This allowed us to immediately determine if our code fulfilled our specification and allow for incremental debugging, which helped us implement changes to our code with more confidence. Using test driven development on our system allowed us to continuously train the model and learn from the data. It also allowed us to gain a deeper understanding of the system. We believe that the process of building the model, training it, and testing the model lead to more accurate predictions than if we did not use test driven development.

For our first model, we wanted to start with a smaller size of input data, which is we built a model to recognize handwritten digits, as there are only ten possible digits. For this model, we were more focused on understanding the convolutional neural network which is why we did not include RNN layers or CTC. We used five epochs for our model, as we believed cycling through the training dataset five times would be enough such that the model no longer improves. Our dataset contained over eighty thousand parameters, which is why we believed five epochs was an acceptable value. At the fifth epoch, we saw our validation accuracy < accuracy. We assumed that since our validaiton and symbol accuracy should be closer, it seemed like our model did well and we didn't overfit our data.

However, we noticed that after three epochs we achieved an accuracy score of 0.9759. The figure below shows the process of fitting our model, and contains the corresponding data for each epoch. The goal of our model was to maximize the accuracy score, as a higher accuracy score (closer to one) implied our model would make more accurate predictions. Since our model achieved a high accuracy score after three epochs, we believe that five was likely too high for the number of epochs. Increasing the number of epochs could lead to overfitting, which would lead to a large gap between training loss and generalization loss. Given that our core vision is to create a model that can be generalizable and accessible, we believe that it is important to take into consideration the number of epochs to ensure that we are not overfitting our model, causing it to be too complex and not recognize new and unseen data. We believe that this is the main reason our model struggled to recognize the digits "4" and "7"—we used two different handwriting styles to write the number but our model was unable to recognize either version. While our model was able to recognize every other digit we still believe the model was overfit which led to a loss of prediction regarding the digit four. To correct our potential overfitting, it would be acceptable to drop a layer.

```
Epoch 1/5
1313/1313 [==============================] - 28s 20ms/step - loss: 0.3378 - accuracy: 0.8936 - val_loss:
0.1213 - val_accuracy: 0.9641
Epoch 2/5
1313/1313 [==============================] - 30s 23ms/step - loss: 0.1044 - accuracy: 0.9674 - val_loss:
0.1012 - val_accuracy: 0.9687
Epoch 3/5
1313/1313 [==============================] - 39s 30ms/step - loss: 0.0739 - accuracy: 0.9777 - val_loss:
0.0711 - val_accuracy: 0.9777
Epoch 4/5
1313/1313 [==============================] - 34s 26ms/step - loss: 0.0593 - accuracy: 0.9812 - val_loss:
0.0615 - val_accuracy: 0.9812
Epoch 5/5
```

```
1313/1313 [==============================] - 30s 23ms/step - loss: 0.0470 - accuracy: 0.9847 - val_loss:
0.0682 - val_accuracy: 0.9795
```

For our second baseline model, we were confident in our ability to implement a model that used CNNs, and wanted to use RNNs and CTC to create a more accurate model that could be trained on more complex data. Given that we wanted to work with characters, we believed adding RNN layers and CTC would allow us to add more input data without compromising accuracy. This is because while there are only ten possible digits in our first model, our second model has seventy nine characters (number of characters in IAM dataset). Moreover, we believe that the model will perform better with the addition of RNN, as our model is sequential (similar to our first model). RNN is best suited for sequential data as it can handle arbitrary input and output lengths and uses its internal memory to process the sequence of inputs. We believed this would improve our model for words as digits have a stationary input and output size (size of one) whereas words have arbitrary sizes, and the sequence of characters in a word is dependent on the preceding and succeeding characters. The addition of CTC allowed us to train our neural network without knowing the alignment between the input and the output, making it ideal for handwritten recognition for words.

As for evaluating our system, we started with a basic test case, to recognize the handwritten word “little” from the sample dataset we used to train and validate our model. As discussed in Results, our model was able to recognize the word with a probability rate of 0.966256. Running the test dataset on our model yielded a character error rate of 10.624916% and a word accuracy rate of 73.686957%. The character error rate implies a character accuracy rate of 89.375084%. While neither word accuracy rate or the character accuracy rate are as high as the accuracy rate from our first model, we believe that this model was still an improvement. The addition of RNN layers and CTC allowed us to work with a more complex dataset. The IAM dataset contains 115,320 words. Given the increase in the size of the dataset, the increase in the size of the inputs themselves, and the complexity of the dataset, we believe these accuracy values still indicate an improvement from our first model, despite the fact that these additions led to lower accuracy values.

As mentioned previously, the goal of our model is accessibility. We hope to create a model that is generalizable. Keeping test driven development in mind, after we were able to create a model that recognized the sample image “little” from the IAM dataset, we moved on to inputting custom data for our model to recognize. We believe it is important to test our model on various handwriting styles and fonts to be able to achieve our goal. As seen in Results, the character error rate is 13.04% and the word accuracy

rate is 50%. The character error rate implies a character accuracy rate of 86.96%. The character accuracy value for our custom data is close to that of testing data, but the word accuracy rate for our custom data is much lower. Compared to our first model, we do not believe we overfitted our model based on the number of epochs. This is because we stopped cycling through the data as soon as we recognized that the model was not improving, ensuring we did not overfit on this basis. However, we do believe that the increase in complexity of our model (5 CNN layers, 2 RNN layers, and CTC) may have caused overfitting and decreased generalizability. While the model is able to recognize a majority of the characters, the probability values corresponding to words are relatively low (around 30%) indicating our model is not confident in its predictions. We also believe that the complexity of the dataset may have contributed to the overfitting of our model. We wanted to create a model that would be able to work with a diverse input dataset, and we do believe that we were successful in this regard, but we also recognize both our models have not achieved the level of generalization we desired.

Conclusion:

One of the key findings from our project was the importance of accounting for a multitude of factors when looking at generalization. Our core vision was to create an Upload and Translate Handwritten Recognition Text model, which we accomplished, but there are still improvements that can be made regarding our accuracy. Our goal was to always create a model that increased accessibility and focused on generalization, and through test driven development we were able to construct our model to read and recognize custom inputs. However, the accuracy is still not as high as we desire, but given our limitations we believe that we achieved our goal within the scope of our time frame and abilities.

The greatest limitations we faced in this project were hardware and time constraints. Our hardware was not prepared to handle the desired size of data we wished to use on our model, and we did not have access to a GPU. These limitations mean we had to constrain the size of our datasets as well as the size of our inputs to ensure our computers had the capacity to train our model. We also had to restart and redo our model several times, part of which was due to the constraints described above. A large part, however, was due to the amount of research we placed into how to best train and create our model. Ultimately, we did not pivot from our core vision of an Upload and Translate Handwriting Recognition Text model because we believed the idea was feasible.

One improvement for future work on our model is enhancing the decoder. We used a greedy algorithm to implement our decoder, and we were still able to successfully decode a large portion of our dataset, we believe that using CTCWordBeamSearch would greatly improve our model. This is because the

CTCWordBeamSearch algorithm uses words constrained by a dictionary, which is extremely helpful when characters within a word are incorrectly recognized. Using a dictionary constrains the output and makes recognition more accurate. Moreover, the algorithm also allows non-word characters such as numbers and special characters, which can be extremely useful and allow our model to translate more diverse inputs.

Another improvement is using text correction. Text correction can be used such that if a recognized word is not contained within a dictionary, we can search for the next best word, or the one most similar to our input. This addition would improve the word accuracy of our model because even if individual characters within a word are not recognized, the algorithm would search for the most similar word, improving the likelihood that the translation of the input image is correct. However, it may also run into issues with non-english names like “oluwatise,” “neha,” and “heysil.”

Initially, we planned on creating a GUI which would allow a user to write text on a digital notepad, and we would have an output panel that would scan the notepad, decode the text, and display the translation. However, we believed it was more important to focus on our model and the AI component than it was to focus on the interactive display. We do believe that the user interface is a very powerful application and a good direction for future improvements, as it once again improves accessibility.

We did follow a tutorial implementation where our GUI could read the handwritten digits on a video. However, the experience showed low accuracy on our side compared to the tutorial demo and required a lot of processing power. Since our hardware constraints limited our ability to run the program multiple times efficiently for testing, we only demoed it once during our final presentation. Regarding our code submission, it’s commented out as we decided not to improve its accuracy. It was difficult to program that experience iteratively and apply test driven development to the model tutorial so we’ve cited said code instead.

References

A. F. de Sousa Neto, B. L. D. Bezerra, A. H. Toselli and E. B. Lima, "HTR-Flor: A Deep Learning System for Offline Handwritten Text Recognition," 2020 33rd SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI), 2020, pp. 54-61, doi: 10.1109/SIBGRAPI51738.2020.00016.

Assogba, Yannick. "Handwritten Digit Recognition With CNNs | TensorFlow.js." *TensorFlow*, 16 May 2022, www.tensorflow.org/js/tutorials/training/handwritten_digit_cnn.

DeepLearning_by_PhDScholar. "Deep Learning- Handwritten Digits Recognition Tutorial | Tensorflow | CNN | for Beginners." *YouTube*, 24 Oct. 2020, www.youtube.com/watch?v=u3FLVbNn9Os.

Flor, Arthur. "A Robust Handwritten Recognition System for Learning on Different Data Restriction Scenarios." *A Robust Handwritten Recognition System for Learning on Different Data Restriction Scenarios - ScienceDirect*, Apr. 2022, https://doi.org/10.1016/j.patrec.2022.04.009.

Notability. "Handwriting Search and Conversion in Notability." *YouTube*, 29 June 2018, www.youtube.com/watch?v=H6d8NF7ZqT8.

Paul, Sayak, and Aaakash Nain. "Keras Documentation: Handwriting Recognition." *Handwriting Recognition*, 16 Aug. 2021, keras.io/examples/vision/handwriting_recognition.

Scheidl, Harald. "Build a Handwritten Text Recognition System Using TensorFlow." *Medium*, 3 May 2021, towardsdatascience.com/build-a-handwritten-text-recognition-system-using-tensorflow-2326a3487cd5.