University of Arkansas – CSCE Department

Capstone II – Final Report– Fall 2021

Deep Learning Handwriting Recognition Model

William Farris, Baron Davis, Micheal Oyenekan, Creighton Young

Abstract

The problem to solve is to improve an open-source handwriting recognition model. The overall objective is to improve the already 75% accuracy to 90% accuracy with several avenues to continue forward. As time goes on, scanning documents for information will get increasingly more important as copying or translating written information to a machine-readable state takes time and money to do. For this reason, it is important for a program to exist that scans documents for letters and words and converts them to a far more readable and easy-to-store state for computers so that chronicling information is faster for people who need to record information but cannot bring devices with them to do so.

1.0 Problem

As we continue to store more data electronically, there are still various documents that are being handwritten, such as loan applications or medical reports. Handwritten reports may not have machine-readable text that document processors can process.

This problem has led to the development of Handwritten Text Recognition (HTR), which applies Deep Learning to process handwritten documents into a machine-readable form. However, open-source HTR projects have yet to reach a level of performance to be used in enterprise applications. Because handwriting differs between each person, so does accuracy. Having a computer be able to read in handwriting regardless of writing style allows data logging individuals to save time when it comes to copying that information and reduce the likelihood of errors, such as hand-typed in typos, in the machine-readable text.

2.0 Objective

The objective of this project is to build upon and experiment with model architectures and data manipulations of an open-source HTR Deep Learning model to improve the accuracy of performance. To improve the accuracy of the HTR Deep Learning Model, we worked with several technologies used in the model, including the CNN layers, RELU function, replacing the

optimizer with an Adam optimizer, and switching between an LSTM and a 2D LSTM, and increasing the dataset size with the IAM.

By experimenting with the model, we were able to gain more experience and understanding of each individual part, and thus applied our newly gained knowledge of machine learning, improving aspects of the model as well as resulting in improved accuracy.

3.0 Background

3.1 Key Concepts

The first key technology that we dealt with was **CNN layers**. CNN is short for convolutional neural network. Using a CNN allows us to use nodes to assign importance, also known as **weight,** and also assigns threshold [7]. CNN is a common approach to dealing with handwritten character recognition and was one of our focus points while experimenting [1]. Our input image would first be fed into our CNN layers. These layers are trained to extract the relevant features from the image that we need. Every layer in the CNN would consist of 3 operations. The first operation would apply a filter kernel of 5x5 in the first two layers and a 3x3 in the last 3 layers of the input. A filter kernel is a matrix of numbers that would be multiplied times our input image in order to alter the image values. After that we would apply a nonlinear RELU, Rectified Linear Unit, function. Finally, a pooling layer summarizes image regions and outputs a downsized version of the image.

The CNN layer uses nodes known as **Neurons** that are placed in layers. The connections neurons have between each other are known as **synapses.** The neurons first receive an input signal from a source, perform calculations, then send output signals further in the CNN through the synapse.

With each synapse there is a **weight** that goes along with it. This weight represents the strengths between the neurons. If the weight from node 1 to node 2 has greater magnitude, it means that neuron 1 has greater influence over neuron 2. A weight decides how much influence the input would have on the output.

Along with the weight there is a corresponding **gradient** that goes along with it. The gradient tells how sensitive the cost is to change in its weight. If we have synapse A that has a gradient of 3.2 and a synapse B that has a weight of .1, then a change in weight of synapse A is 32x more sensitive than a change in the weight of synapse B.

After our input is run through the CNN, a nonlinear **RELU function** is applied to it. RELU, Rectified Linear Unit, is a piecewise function that would return our input if our input is greater than zero, and would return 0 if our input is less than 0. The RELU function is linear for values greater than zero which makes it great for **backpropagation**.

Backpropagation is a class of algorithms that compute the gradient of the loss function with respect to its weights. Backpropagation is how we adjusted the weights and biases of our

neural network in order to get the desired neurons to fire. For example, if we are trying to have the letter A be recognized by our neural network, we would want to adjust our values and biases so that neurons that have a positive impact on our neural network recognizing the letter A become more active.

RELU functions are often used because it is a model that is easier to train and often has satisfactory performance [2]. We applied this nonlinear RELU function because it would be close to linear or linearly separable. To understand **linear separability,** imagine there is a set of data points with half being blue and half being red. The data set would be linearly separable if there exists a line that can be drawn in the plane such that all the blue points are on one side of the line and all the red points are on the other side.

The other reason a RELU function is used, as opposed to a sigmoid function, which is a function that would return values ranging from either 0.0 to 1.0., is that the RELU function does not suffer from the **vanishing gradient problem**. The vanishing gradient problem, also known as the exploding gradient problem, is a problem that arises when computing gradients during backpropagation. In order to calculate the gradient for a node in a neural network, we used all node gradients that have outward synapses to our current node. There is no issue at the start, but as we go further back each node starts to have more and more connecting gradients, resulting in an increasing number of calculations. With how large our neural network is, this problem gets worse as worse as there are tons of nodes. It gets worse and worse due to a gradient at any point, being the multiplication of all gradients at prior layers. If we were to use a sigmoid function, all our gradients would be between 0 and 1, which would mean as backpropagation occurs, each nodes gradient gets smaller and smaller. As a result of these small numbers, our accuracy would be very low, along with backpropagation would take a long time. Therefore, a RELU function is used, as it is not bounded by 0-1, meaning no vanishing gradient.

RNN, Recurrent Neural Network, are a class of artificial neural networks where connections between nodes form a directed graph along a temporal sequence. RNNs would allow us to use previous outputs to be used as inputs, so for ours we would be using the outputs from our CNN to pass them into our RNN. One problem with RNNs is that they too can also often lead to a vanishing/exploding gradient. A special type of RNN called **Long Short-Term Memory (LSTM)** can also help to solve this issue.

An **LSTM** is a special type of RNN. LTSMs specialize in remembering information for long periods of time. They can propagate information over longer distances and provide more robust training characteristics than normal RNN. The reason why LSTMs can help with solving the vanishing gradient problem is through its use of a cell state and a forget gate. The reason for this is very math heavy, but they effectively solve the vanishing gradient problem by preventing the gradient from going to 0 as a function of the number of samples seen thus far [16]. Our model consisted of two LSTM layers.

Another key technology that we used was a **2D-LSTM**. A 2D-LSTM is an alternative kind of LSTM. The difference between a 2D LSTM and a regular LSTM is that a 2D LSTM takes in 2D input as its input, as opposed to the regular 1D input. A 2D LSTM was a potential

improvement in this project due to the fact the input would be passing in is an image, which came in a 2D form. Although we were not sure if this would improve the accuracy to start [3], it was one of the focal points of our experimentation.

After our image goes through the LSTM, that output along with the ground text is sent to a **CTC**. CTC stands for Connectionist temporal classification and is a type of neural network output and scoring function that is used for training LSTMs. The CTC function would compute the loss value of the neural network. The loss value is basically how well or how poorly our neural network was doing. The higher the loss value the worse that the neural network was doing. A perfect neural network would have a loss value of zero.

In order to train our dataset, we used an input data set from **IAM**. The IAM Handwriting Database contains forms of handwritten English text which can be used to train and test handwritten text recognizers and to perform writer identification and verification experiments. The IAM database consists of 1,539 pages of scanned text, 5,685 isolated sentences, 13,353 lines of text, and finally 115,620 words.

Error in machine learning is known as **Loss.** Loss can occur to due to a variety of reasons, including inaccurate assigned weights. This is especially the case due to how weights are imperfect and not always accurate. In order to deal with loss, we used an **optimization algorithm** to update network weights and the learning curve of the neural network, allowing for more accurate results.

Another term/concept that would be involved in this project is an **Adam optimizer**. An Adam optimizer is an optimization algorithm that can be used to iteratively update network weights based on training data. The Adam optimizer is used instead of a classical stochastic gradient descent procedure. A stochastic gradient descent would maintain the single learning rate, alpha, for all weight updates and the learning rate does not change during training. Adam instead calculates an exponential moving average of the gradient and the squared gradient, and the parameters beta1 and beta2 control the decay rates of these moving averages. Beta1 is the exponential decay rate for the first moment estimates, while beta2 is the exponential decay rate for the second-moment estimates. The downside of using Adam was that our number of epochs increased, thus resulting in longer periods of time training the model.

For our project we often got various inputs from letters that looked very similar to each other. Examples of characters that were easily mistaken for one another are the characters “a” and “o” and the characters “l” and “i”. In order to distinguish between these characters, we looked at a very small feature of each character that can be much harder for a machine to find. We attempted to fix this by using decoder with one of two algorithms. One of these two algorithms is word beam search. **Word beam search** worked by having each of our inputs be in either two states, a word state or a non-word state. When in the word state we were only allowed characters that would form words, compared to when we were in a non-word state, we did allow characters like “ “. We could only move from a word state to a non-word state when we are finished with a word and could move from the non-word state to the word state when we receive another character.

The other algorithm that we could use was called **token passing**. For token passing we used a dictionary and a word language model. The algorithm would search for the most probable sequence of words in the dictionary in the neural network output. One problem with token passing is that it could struggle with punctuation in words and numbers.

We have also used a **python spellchecker** to influence our model. A spellchecker allows for text to be exposed to the algorithm, and is capable of detecting, highlighting, and correcting grammatical errors within said text. This can be implemented within our program to detect possible grammatical errors that result due to incorrectly read text within the model.

3.2 Related Work

CAPTCHA [11] is a service that checks if the visitor is a robot by sending them a garbled message that only humans can solve while computers struggle with immensely. ReCAPTCHA on the other hand decides to take this pattern recognition ability that human brains have and put it towards digitizing words from printed media and help train AI to recognize images in a photo. Our project was meant to work on its own after a good amount of training and it was also open source in comparison to ReCAPTCHA.

Gboard [12] is a service provided by google that incorporates both predictive text and translating handwriting to text. However, it is only available on Android devices and backed by Google while our handwritten recognition model would be open source and thus free to use elsewhere.

Microsoft OneNote [13] is a service that is offered by Microsoft that not only accepts PDFs, but it can also translate any written text on it into machine-readable text. This may seem like a superior program to our project, but, however, it’s run by Microsoft, while ours was open source, making it more available for others to use as a basis in their own programs.

4.0 Design

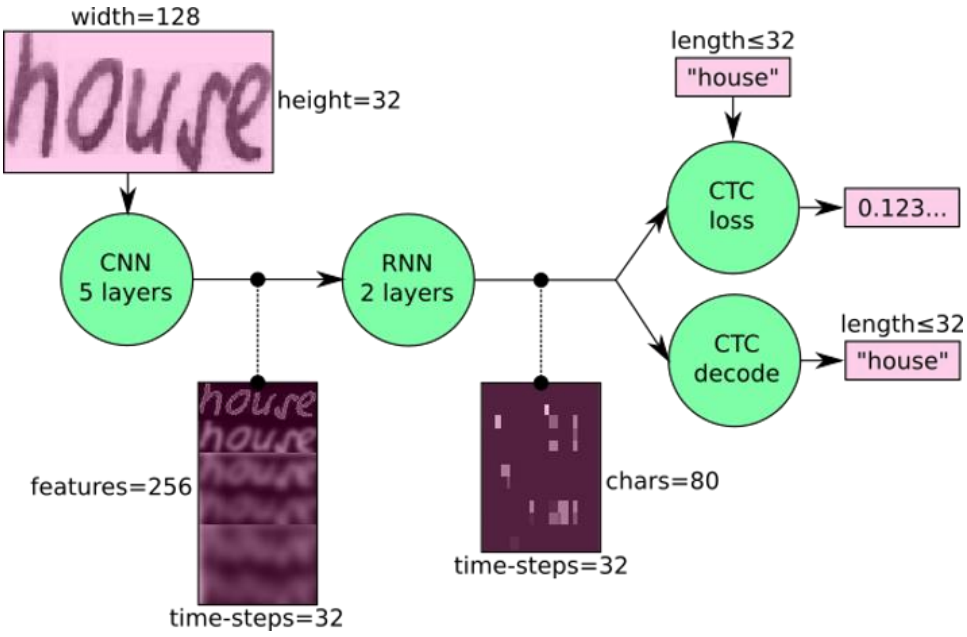
4.1 Requirements and/or Use Cases and/or Design Goals

Our primary design goal was to achieve 90% accuracy from the initial 75% accuracy from the given open-source HTR Deep Learning model. We would experiment with the code from a variety of approaches to attempt to achieve this 90% accuracy.

4.2 Detailed Architecture

The primary focus in our project is making improvements to each aspect of the overall model architecture. One of the aspects of the model we were working on is the input images. By changing the input dataset, we can change the accuracy to which the model is able to discern each word.

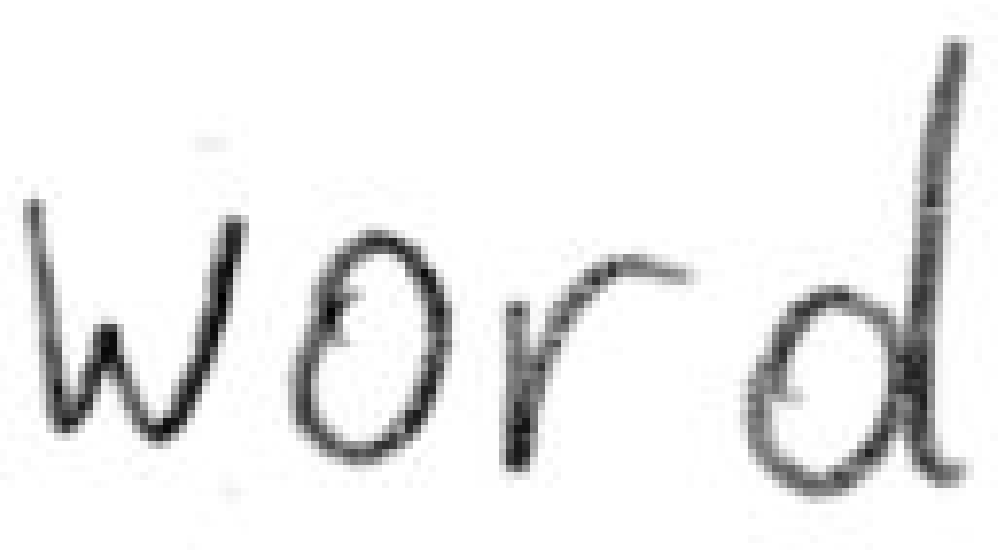
Below is a simple overview of the model [15]



The model used 5 CNN layers that allow for the model to identify features within the input image. These features allowed for the model to distinguish between individual characters. More features allowed for the model to detect more individual features of a character, but having too many features could cause the model to detect unexpected features that can harm the accuracy of the model. As such, we determined that changing the number of CNN layers had lower priority than other approaches.

There were 2 RNN layers that allowed the model to make connections and produce predicted results. These results are then passed to the decoder, which makes use of the information to determine the produce a guess for the word along with outputting a confidence level.

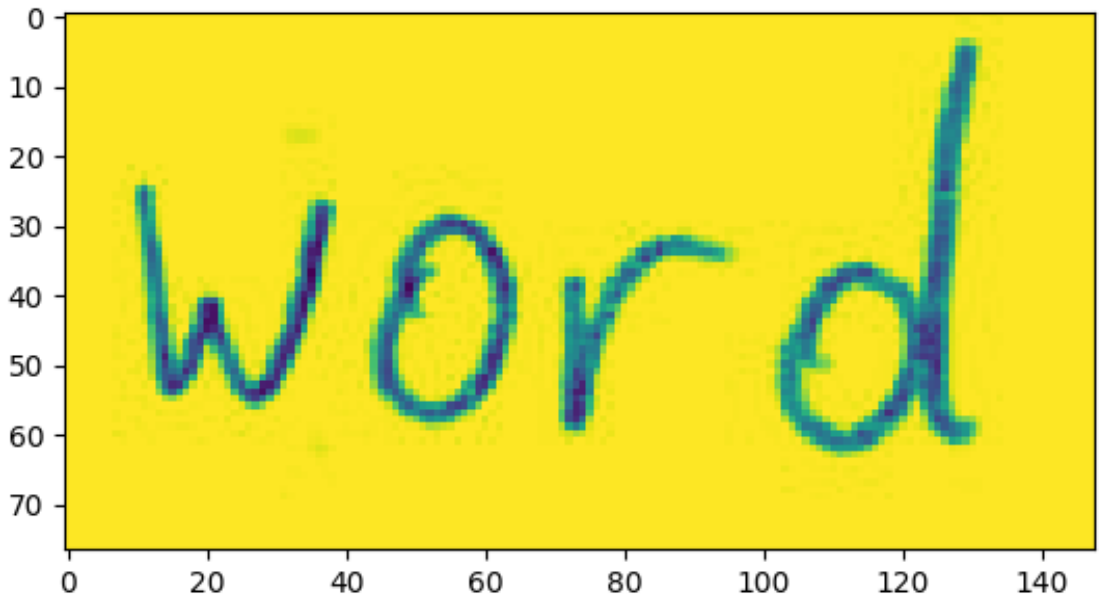
Below is an example of one of our input words:



The above word's recognized probability:

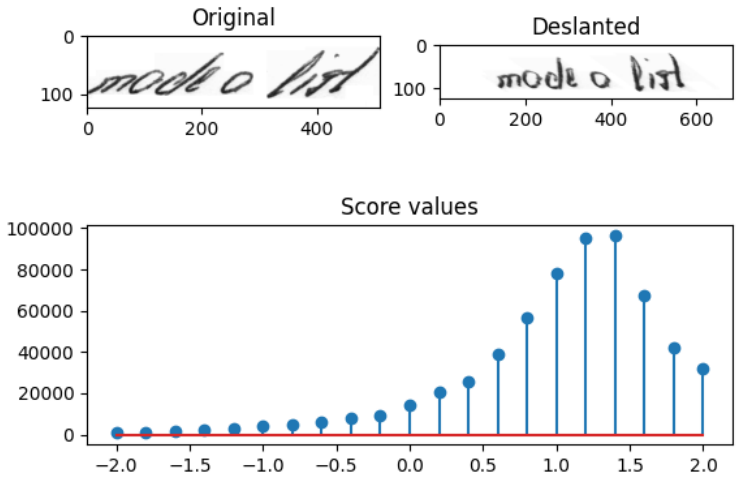
```
Recognized: "word"
Probability: 0.9513835310935974
```

We tried utilizing DeslantImg to process the word and remove any slanted writing style, customary to cursive, and then funnel it into the process. Below is an example word that had been utilized in this fashion and it neatly outputted the processed sample. However, due to how our current sample given to us did not have any slant, it did not change with the word at all.

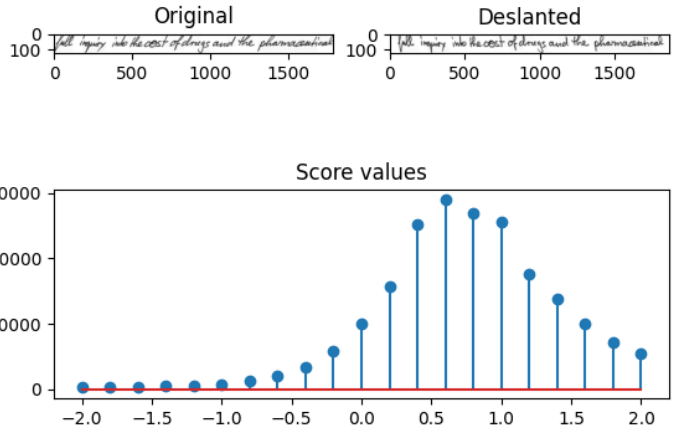


In the Deslant program, we were given example images to process through the program and we have been given these results.

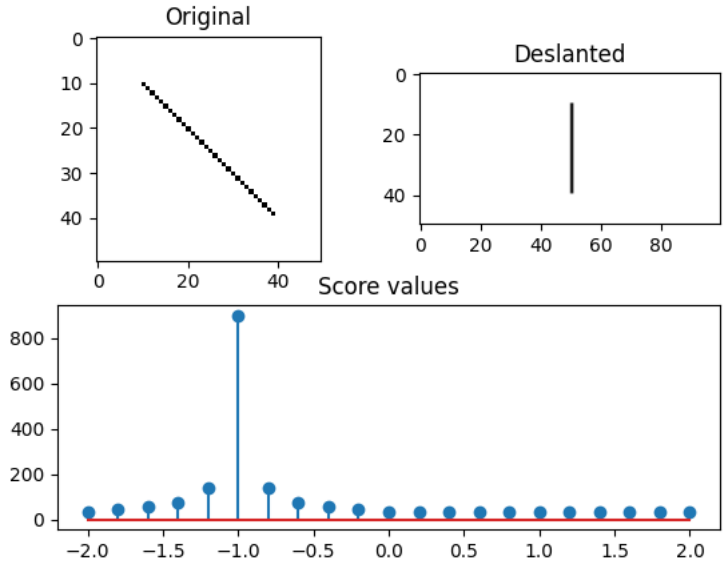
These were tests 1 through 5, given to use by the DeslantImg github and practicing processing it. Test 1 was the standard test, just a simple, short phrase that's easily readable.



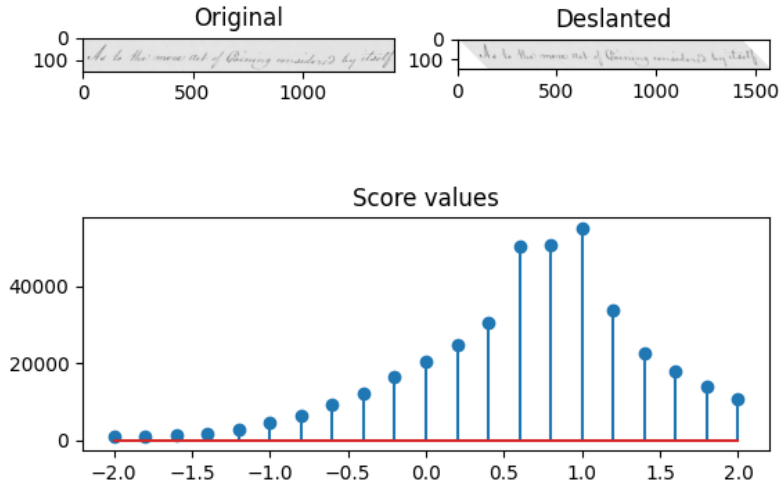
Test 2 was a bit of a longer phrase or sentence to smoothen out.



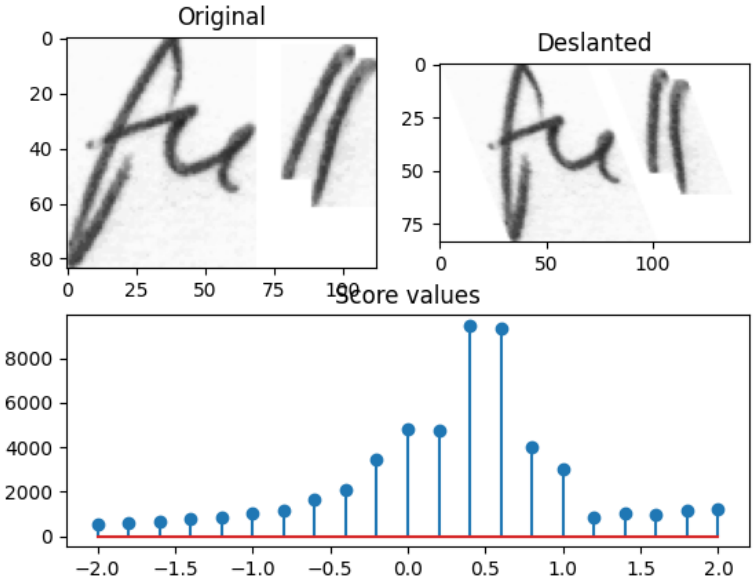
Test 3 was a test of minimalism, taking a single line and deslanting.



Test 4 was long and very faint, testing if it was readable enough for the program to work.



Test 5 was short and bold, even a little blurry and different, to see if it worked here too with a drastically different style.



We also experimented on the dataset upon which the model had been trained. For this, we used an IAM dataset which contained a database of many different handwritten words.

Below is one of the words in our IAM database:

We also added an augmented dataset based on the IAM dataset that takes the images and applies changes to them, such as blurring the image, applying random transformations, adjusting line thickness, and adding random noise.

Below is an example of an image with increased blur:

As we experimented on our project, we have also focused on other approaches to our model.

One such example was using our model on full lines, rather than single word inputs.

For example, we would use the following:

Our most successful approach was through the postprocessing of the model's output, specifically through spelling correction. The packages we originally tried for our spelling correction were pyspellchecker and textblob. After testing how successful each one was, we found that pyspellchcker led to an increase of about 8% in the model, while textblob only increased the model by about 5%. Along with this we also tried to implement grammar correction to subsidize the spell correction. The original intent was, as pyspellchecker gives you a list of possible candidates for correction, to send all of the possible corrections, in sentence form, to the grammar corrector, and test all potential candidates to see which sentence was the most grammatically correct. However, this idea had to be scrapped as all the grammar packages we could find, gramformer, languagetool, and grammar-check, either were broken or did not give us an output that would be helpful for the model.

By implementing the spellchecker with our other approaches, we determined that deslantimg was causing harm to our results and caused an approximate 5% decrease in accuracy. Certain aspects of our augmented dataset were also able to provide improvements, notably the

images with random noise and the randomly stretched images. Overall, we were able to increase the accuracy of the model from approximately 70% to 81%. These numbers can vary and are our approximate results after calculating the accuracy multiple times.

4.3 Risks

Risk	Risk Reduction
Incorrect output poses a risk to the data and thus can cause issues in recording	By raising the accuracy of the output, we can reduce the risk of errors.
Unique handwriting styles or sloppy handwriting may influence the output	Working with a variety of input and working with the de-slant image code may reduce this risk

4.4

We were provided with a repo that already contained prebuilt methods and command line arguments for training and validating the model. Thus, most of our time was spent on experimenting with different model architectures and image transformation methods rather than building the train/test pipeline from scratch. The original model performed with an accuracy of around 75% on the IAM Offline Handwritten Text Recognition (HTR) dataset. We experimented to improve the performance of the Open-source Handwritten Text Recognition (HTR) Deep learning model to meet the enterprise applications performance threshold of ~90%.

For implementation and design, the model that was offered to us was an open-source code in a Github repository. This Handwritten text recognition model used Python as the programming language, implemented with TensorFlow (TF) and trained on the IAM offline HTR dataset. The model took images of single words or text lines (multiple words) as input and output the recognized text. Three quarters of the words from the validation-set were correctly recognized, and the character error rate was around 10%.

We used an agile methodology to manage our project and the framework we chose was bi-weekly scrum. The timeline was between 16 – 20 weeks (about 4 and a half months). CGI acceptance rate for projects is 90%. So, in order to get our project to 90% we met meeting every two weeks and assigned tasks by sprint.

SPRINT 1

- Create Website
- Create Github Account
- Setup Jira Board to assign tasks

- Fork repository to our team repository
- Create virtual environment to add all plugins and packages

SPRINT 2

- Research Spike: Data Augmentation Methods for handwriting
- Load IAM Dataset
- Develop Data Augmentation Python class- We used the information from Research Sike to develop this class
- Creating the Augmented Dataset- We used the data set that was provided by [17]. The dataset characteristics have 657 writers who contributed samples their handwriting
- Data augmentation: As a result of limited data, we would increase the dataset-size by applying further (random) transformations to the IAM input images. These changes would be flipping, translations, or rotations to our datasets. By creating these changes our neutral network created a different dataset. Now, only random distortions are performed. The four augmentations we implemented were random noise, blur, random stretching of the image, and adjusting the line thickness

SPRINT 3

- Train and Validate New Model on Augmented Dataset and put together result
- The Decoder- It uses token passing or word beam search decoding to constrain the output to dictionary words. We were able to add this to features.
- Change the Optimizer
- Remove cursive writing style in the input images (see DeslantImg). We will be using the **Deslanting Algorithm**, which can be found in an open-source repository. This algorithm sets handwritten text in images upright, i.e., it removes the cursive writing style. One can use it as a preprocessing step for handwritten text recognition.

SPRINT 4

- Integrate and experiment with spellcheckers- We decided to add more spell checkers to improve our algorithm. We added Py spellchecker and Text blob to help correct the model's mistakes. We tried to use Gingerit grammar checker to check the grammar of the output but failed after several tries.
- Text correction- Often when our program is trying to decode the handwriting, it is often off by one or two letters. For example, it could possibly read "oronge" when the real word is supposed orange. The text correction, using a set dictionary of common words, we will try different combinations of commonly confused letters, like a's/o's m's/n's r's/n's.
- Increase CNN layers
- Replace **LSTM** by **2D-LSTM**- We were unable to add this feature because it will cause algorithm to be overfitted

4.5 Schedule –

Tasks	Dates
1. Meeting with Industry champion on what needs to be done…	10/19
2. Get familiar with repo, AI, and python	10/26-11/30
3. Data augmentation	11/16 - 11/30
4. Increase input size	12/14 - 1/25
5. Load IAM Dataset Research Spike: Data Augmentation Methods for handwriting Develop Data Augmentation Python class Creating the Augmented Dataset	1/25 - 2/17
6. Replace optimizer Remove Cursive Writing Style Train and Validate New Model on Augmented Dataset The Decoder	2/22 - 3/15
8. Text correction Add more CNN layers Replace LSTM by 2D-LSTM Decoder.	3/15 - 3/29
9. Testing and Validation	3/29 - 4/19
10. Documentation	4/19- 5/3

4.6 Deliverables

- Design Document: Contains a listing of each major software component and the resulting changes to each component
- Initial data: The starter code: SimpleHTR, DeslantImg, LineHTR, CTCWordBeamSearch
- Python code for the resulting HTR program
- Final Report

5.0 Key Personnel

William Farris is a Senior Computer Science major in the Computer Science and Computer Engineering Department at the University of Arkansas. He has completed relevant courses.

Baron Davis is a senior Computer Engineering major in the Computer Science and Computer Engineering Department at the University of Arkansas. He has completed relevant courses.

Creighton Young is a senior Computer Science major in the Computer Science and Computer Engineering department at the University of Arkansas. He has completed relevant courses.

Micheal Oyenekan is a senior Computer Engineering major in the Computer Science and Computer Engineering department at the University of Arkansas. He has completed relevant courses.

Nathaniel Zinda is a Machine Learning Engineer at CGI and was the main contact for the first half of this project.

Rishi Dhaka is a Machine Learning Engineer at CGI and was the main contact for the remaining portion of the project.

6.0 Facilities and Equipment

Due to the nature of the project at hand, the equipment and equipment. On the equipment end, personal computers were all that were necessary to use. For facilities, computer labs would occasionally be useful in the case of meetings and working together.

7.0 References

[1] D. S. Maitra, U. Bhattacharya and S. K. Parui, "CNN based common approach to handwritten character recognition of multiple scripts," 2015 13th International Conference on Document Analysis and Recognition (ICDAR), 2015, pp. 1021-1025, doi: 10.1109/ICDAR.2015.7333916.

[2] A Gentle Introduction to the Rectified Linear Unit (ReLU), https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/

[3] Mohammad Reza Yousefi, Mohammad Reza Soheili, Thomas M. Breuel, Didier Stricker, "A comparison of 1D and 2D LSTM architectures for the recognition of handwritten Arabic," Proc. SPIE 9402, Document Recognition and Retrieval XXII, 94020H (8 February 2015)

[4] Research Group on Computer Vision and Artificial Intelligence — Computer Vision and Artificial Intelligence. (2021). Retrieved 5 November 2021, from https://fki.tic.heia-fr.ch/databases/iam-handwriting-database

[5] Brownlee, J. (2017). Gentle Introduction to the Adam Optimization Algorithm for Deep Learning. Retrieved 5 November 2021, from https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

[6] Word Beam Search: A CTC Decoding Algorithm. (2020). Retrieved 5 November 2021, from https://towardsdatascience.com/word-beam-search-a-ctc-decoding-algorithm-b051d28f3d2e

[7] Convolutional Neural Networks, https://www.ibm.com/cloud/learn/convolutional-neural-networks

[8] Various Optimization Algorithms For Training Neural Network, https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6#:~:text=Optimizers%20are%20algorithms%20or%20methods,help%20to%20get%20results%20faster

[9] Deep Learning Neural Networks Explained in Plain English, https://www.freecodecamp.org/news/deep-learning-neural-networks-explained-in-plain-english/

[10] Brownlee, J. (2019). How to Fix the Vanishing Gradients Problem Using the ReLU. Retrieved 29 November 2021, from https://machinelearningmastery.com/how-to-fix-vanishing-gradients-using-the-rectified-linear-activation-function/

[11] Web Security Words Help Digitize Old Books, https://www.npr.org/2008/08/14/93605988/web-security-words-help-digitize-old-books

[12] Gboard, https://en.wikipedia.org/wiki/Gboard

[13] Microsoft OneNote, https://en.wikipedia.org/wiki/Microsoft_OneNote

[14] Data Augmentation, https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/

[15] Build a Handwritten Text Recognition System using TensorFlow. (2021). Retrieved 30 November 2021, from https://towardsdatascience.com/build-a-handwritten-text-recognition-system-using-tensorflow-2326a3487cd5

[16] Recurrent Neural Networks, the Vanishing Gradient Problem, and LSTMs
Recurrent Neural Networks, the Vanishing Gradient Problem, and LSTMs. (2019). Retrieved 30 November 2021, from https://medium.com/@pranavp802/recurrent-neural-networks-the-vanishing-gradient-problem-and-lstms-3ac0ad8aff10

[17] IAM Dataset, http://www.fki.inf.unibe.ch/databases/iam-handwriting-database