Handwritten Text Recognition using Deep Learning

Batuhan Balci
bbalci@stanford.edu

Dan Saadati
dans2@stanford.edu

Dan Shiferaw
shiferaw@stanford.edu

1. Abstract

This project seeks to classify an individual handwritten word so that handwritten text can be translated to a digital form. We used two main approaches to accomplish this task: classifying words directly and character segmentation. For the former, we use Convolutional Neural Network (CNN) with various architectures to train a model that can accurately classify words. For the latter, we use Long Short Term Memory networks (LSTM) with convolution to construct bounding boxes for each character. We then pass the segmented characters to a CNN for classification, and then reconstruct each word according to the results of classification and segmentation.

2. Introduction

Despite the abundance of technological writing tools, many people still choose to take their notes traditionally: with pen and paper. However, there are drawbacks to handwriting text. It's difficult to store and access physical documents in an efficient manner, search through them efficiently and to share them with others.

Thus, a lot of important knowledge gets lost or does not get reviewed because of the fact that documents never get transferred to digital format. We have thus decided to tackle this problem in our project because we believe the significantly greater ease of management of digital text compared to written text will help people more effectively access, search, share, and analyze their records, while still allowing them to use their preferred writing method.

The aim of this project is to further explore the task of classifying handwritten text and to convert handwritten text into the digital format. Handwritten text is a very general term, and we wanted to narrow down the scope of the project by specifying the meaning of handwritten text for our purposes. In this project, we took on the challenge of classifying the image of any handwritten word, which might be of the form of cursive or block writing. This project can be combined with algorithms that segment the word images in a given line image, which can in turn be combined with algorithms that segment the line images in a given image of a whole handwritten page. With these added layers, our project can take the form of a deliverable that would be used by an end user, and would be a fully functional model that would help the user solve the problem of converting handwritten documents into digital format, by prompting the user to take a picture of a page of notes. Note that even though there needs to be some added layers on top of our model to create a fully functional deliverable for an end user, we believe that the most interesting and challenging part of this problem is the classification part, which is why we decided to tackle that instead of segmentation of lines into words, documents into lines, etc.

We approach this problem with complete word images because CNNs tend to work better on raw input pixels rather than features or parts of an image [4]. Given our findings using entire word images, we sought improvement by extracting characters from each word image and then classifying each character independently to reconstruct a whole word. In summary, in both of our techniques, our models take in an image of a word and output the name of the word.

3. Related Work

3.1. Early Scanners

The first driving force behind handwritten text classification was for digit classification for postal mail. Jacob Rabinows early postal readers incorporated scanning equipment and hardwired logic to recognize mono-spaced fonts [3]. Allum et. al improved this by making a sophisticated scanner which allowed for more variations in how the text was written as well as encoding the information onto a barcode that was printed directly on the letter [4].

3.2. To the digital age

The first prominent piece of OCR software was invented by Ray Kurzweil in 1974 as the software allowed for recognition for any font [5]. This software used a more developed use of the matrix method (pattern matching). Essentially, this would compare bitmaps of the template character with the bitmaps of the read character and would compare them to determine which character it most closely matched with. The downside was this software was sensitive to variations in sizing and the distinctions between each individuals way of writing.

To improve on the templating, OCR software began using feature extraction rather than templating. For each character, software would look for features like projection histograms, zoning, and geometric moments [6].

3.3. Machine Learning

Lecun et. al focused on using gradient-based learning techniques using multi-module machine learning models, a precursor to some of the initial end-to-end modern deep learning models [12].

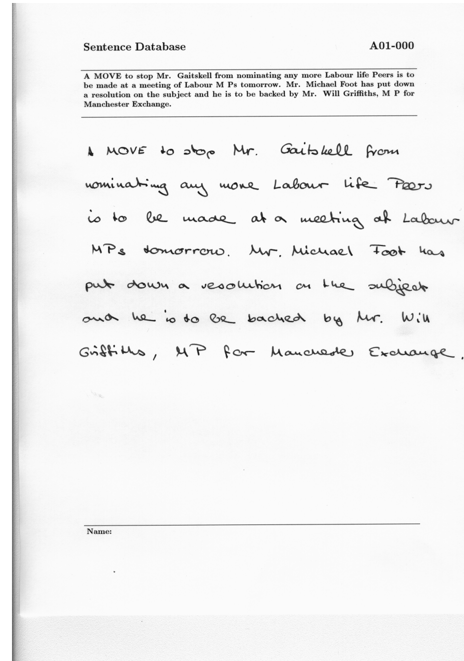
The next major upgrade in producing high OCR accuracies was the use of a Hidden Markov Model for the task of OCR. This approach uses letters as a state, which then allows for the context of the character to be accounted for when determining the next hidden variable [8]. This lead to higher accuracy compared to both feature extraction techniques and the Naive Bayes approach [7]. The main drawback was still the manual extraction features, which requires prior knowledge of the language and was not particularly robust to the diversity and complexity of handwriting.

Ng et. al applied CNNs to the problem of taking text found in the wild (signs, written, etc) and identified text within the image by using a sliding window. The sliding window moves across the image to find a potential instance of a character being present. A CNN with two convolutional layers, two average pooling layers, and a fully connected layer was used to classify each character [11].

One of the most prominent papers for the task of handwritten text recognition is Scan, Attend, and Read: End-to-End Handwritten Paragraph Recognition with MDLSTM Attention [16]. The approach was to take an LSTM layer for each scanning direction and encode the raw image data to a feature map. The model would then use attention to emphasize certain feature maps over others. After the attention map was constructed, it would be fed into the decoder which would predict the character given the current image summary and state. This approach was quite novel because it did not decouple the segmentation and classification processes as it did both within the same model [16]. The downside of this model is that it doesnt incorporate a language model to generate the sequence of characters and words. It is completely dependent on the visual classification of each character without considering the context of the constructed word.

We found a previous CS 231N project to be helpful in guiding us with our task as well. Yan uses the Faster R-CNN model [10] to identify individual characters within a word and for classification. This uses a sliding window across the image to first determine whether an object exists within the boundaries. That bounded image is then classified to its corresponding character. Yan also implements edit distance which allows for making modifications to the classified word to determine if another classified word is more likely to be correct (for instance xoo vs zoo) [9].

4. Data


Sentence Database A01-000

A MOVE to stop Mr. Gaitskell from nominating any more Labour life Peers is to be made at a meeting of Labour M Ps tomorrow. Mr. Michael Foot has put down a resolution on the subject and he is to be backed by Mr. Will Griffiths, M P for Manchester Exchange.

A MOVE to stop Mr. Gaitskell from
nominating any more Labour life Peers
is to be made at a meeting of Labour
MPs tomorrow. Mr. Michael Foot has
put down a resolution on the subject
and he is to be backed by Mr. Will
Griffiths, MP for Manchester Exchange.

Name:

Figure 1. An example form from the IAM Handwriting dataset. Word images in the dataset were extracted from such forms.

Our main resource for training our handwriting recognizer was the IAM Handwriting Dataset [18]. This dataset contains handwritten text of over 1500 forms, where a form is a paper with lines of texts, from over 600 writers, contributing to 5500+ sentences and 11500+ words. The words were then segmented and manually verified; all associated form label metadata is provided in associated XML files. The source text was based on the Lancaster-Oslo/Bergen (LOB) corpus, which contains texts of full English sentences with a total of over 1 million words. The database also includes 1,066 forms produced by approximately 400 different writers. This database given its breadth, depth, and quality tends to serve as the basis for many handwriting recognition tasks and for those reasons motivated our choice of the IAM Handwriting Dataset as the source of our training, validation, and test data for our models. Last but not least, in deep learning large datasets–even with many pre-trained models–are very important and this dataset containing over 100K+ word instances met those requirements (deep learning model need at least $10^5 - 10^6$ training examples in order to be in position to perform well, notwithstanding transfer learning).

4.1. Preprocessing & Data Augmentation

Before training our models with the dataset, we have applied various preprocessing and data augmentation techniques on our dataset in order to make our data more compatible with the models and to make our dataset more robust to real life situations.

4.2. Padding images

As mentioned above, the dataset consists of images of single words only. Moreover, the images are of different sizes because different words are of different lengths and heights. For instance the image of the word 'error' has a lower width than the image of the word 'congratulations' because of the length of the words. Similarly, the image heights differed among images due to the heights of their characters. For instance, the image of the word car has a lower height than the image of the buy since the characters of the word 'buy'extend above and below with 'b' and 'y'. Our architectures, however, assumed the input images to be of the same size just like any other convolutional neural network architecture. This is essential as the weights of the layers are adjusted according to the first input image, and the model would not work as well if weights were not consistent, or changed shapes, for different inputs. Thus, we decided to make all the images of the same shape. It did not make sense to crop large images to an average size because cropping removes some characters from the image, which in turn can cause the image of a word to be interpreted differently. For instance cropping the image of the word 'scholarly' can make the image look like it is the image of the word scholar, and having two different labels for two similar images would definitely negatively affect the accuracy of our models. Moreover, we also thought that rescaling the image size would not work because of a few reasons. First of all, the aspect ratio of every image is different. This means that if we wanted to set the height or the width of an image to a specific value, the other dimension would have been different for all the other images with different aspect ratios. Secondly, if we resized the image on the height and width dimensions independently, then the image would get distorted, and we thought that this again would negatively affect our model since the inherent characteristics of the picture are being lost. Therefore, we decided to pad our images with whitespace to the maximum width and height present in our dataset. While doing so, the white space was added evenly on both sides of the height and the width dimensions. This approach does not necessarily change the inherent characteristic of the word image, and since the images of the same words would pretty much be padded with similar sizes, the relative relationship between the images do not change.

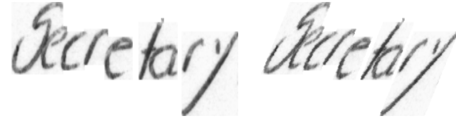


Figure 2. We apply slight rotations to the original image and add that to the dataset with a random probability to model writing in real life that is not perfectly straight

4.3. Rotating images

Even though our dataset consists of the images of every word separately, some words within these images were slightly tilted. This was because the participants of the dataset were asked to write on blank paper with no lines, and some of the words were written in a more tilted fashion. This occasion happens very frequently in real life whether or not the page has lines, thus we decided to make our training data more robust to this issue by rotating an image towards the right by a very small angle with random probability and adding that image to our training set. This data augmentation technique helped us make our model more robust to some minor yet so frequent details that might come come up in our test set.

4.4. Zero-centering Data

$$X^{(k)}_{centered} = X^{(k)} - \frac{1}{N}\sum_{k=1}^{N}\frac{1}{H*W}\sum_{i=1}^{H}\sum_{j=1}^{W}X^{(k)}_{i,j}$$

We center our data by subtracting the dataset mean pixel values from the pixel values of the images before training. This is essential because we use one single learning rate when we update our weights, we want to have a relative scale of the weights among each other so that when we multiply with the single learning rate, all the weights are affected by this multiplication in the same relative manner. If we had not done centering, some weights would be hugely amplified at every update, and others would stay very low.

5. Methods

5.1. Vocabularly Size

As it can be seen in the Data section of our paper, we had a large number of unique words in our dataset. However, some word images appeared in our dataset only a few times, which made it very hard for us to train on these images. This issue, along with the fact that our dataset already had a large vocabulary, encouraged us to cut off some of the words in our dataset and not include those words in our training/validation/test dataset that we were going to use with our models. We therefore limited our data with word images that appeared at least 20 times in our dataset (before splitting into train and validation sets). However, still then

we had around 4000 unique words with at least 20 occurrences in our dataset. To speed our training process up, we decided to limit our vocabulary size significantly down to 50 words, which still took 5-6 hours for training and validation. Our models and algorithms are not dependent on hardcoded number of images and would thus have worked with any number of examples, but we decided to narrow the number of words down for efficiency purposes.

5.2. Word-Level Classification

For our word-level classification model, we first constructed a vocabulary based on randomly selecting 50 words with at least 20 occurrences in our dataset. We trained our word classifier with multiple CNN architectures: VGG-19, RESNET-18, and RESNET-34. The VGG convolutional network architecture was one of the first very deep neural nets to achieve state-of-the-art results on key deep learning tasks. Defying the standard practice at the time, VGG employed much smaller convolutional filters (3 X 3) and a fewer number of receptive field channels in exchange for increasing depth in their networks to balance computational costs for the ImageNet 2014 Challenge. By moving from the traditional 3-7 layers of previous CNNs to 16-19 layers for different iterations of their model, their model not only obtained first and the second places in the localization and classification tracks, respectively, but also was found to generalize well to other computer vision tasks [9]. However, soon after, Residual Networks (RESNETs) topped VGG in first place at the ImageNet challenge[20]. Recognizing that very deep learning networks were difficult to train, in part because of the stagnation of gradient flow to earlier layers in the network. He et. al developed the notion of the residual layer, which instead learned difference functions with respect to the inputs to the layers of the function. Therefore, a layer in a very deep residual network would have the option of learning a zero residual (this tendency can be incrementally enforced with regularization) and thus preserving the input and, during backpropagation, preserving the gradient to earlier layers. This formulation allowed RESNETs with 100+ layers to train comparably in terms of efficiency and parametrization as previous deep learning models and with many more layers. RESNETs are as of the date of this paper one of the most likely candidate cutting-edge deep learning models attempted for computer vision projects.

5.3. Training

We trained all of our models with the ADAM Optimizer on a cross-entropy loss function after softmax activation. For the VGG and RESNET models, we otherwise maintained the same architectures except for altering the last fully connected output layer to instead map to the number of classes (word/character vocabulary). We trained the word and character classifier from scratch. For the character segmentation model, we used a fine-tuned Tesseract model.

Figure 3. These are the filters from the first convolutional layer of the RESNET-18 word segmentation model with the example word "much"

Softmax Loss:

$$L_i = -log(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}})$$

5.4. Segmentation

For word-level classification, we suspected that our performance was suffering because of the large softmax layer output size (there were over 10,000 words in our training vocabulary and well over 100,000 words in the English language alone) and the difficulty of fine-grained recognition of images of words. We decided that character-level classification may be a more promising approach because a fairly comprehensive character vocabulary is relatively much smaller than a similarly comprehensive word vocabulary (the characters A-Za-z0-9 are only 62 distinct symbols), significantly limiting the computational complexity of the softmax. Furthermore, recognition of a character in an image is a simpler task than recognition of a word in an image because of the limited range of characters. However, the first main challenge we would have to encounter in order to test this approach would be segmentation of word images into their component character images. The second main challenge would be recognizing word breaks in image and stringing together consecutive recognized characters in between these word breaks to form words. We will address the earlier of these challenges in this section. In order to implement this task, we employed the new Tesseract 4.0 neural network-based CNN/LSTM engine[13]. This model is configured as a textline recognizer originally developed by HP and now maintained by Google that can recognize more than 100 languages out of the box. For Latin-based languages, including English, the Tesseract model had been trained on about 400000 textlines spanning about 4500 fonts. We then finetuned the parameters of this pretrained model on our IAM handwriting dataset. After we finetuned the model, we segmented the original word input images into their hypothesized component character images, feeding in these output segmented character images into our character-level classification.

5.5. Character-Level Classification

The character-level classification model was quite similar to the word-level classification model. The main differences included passing in character images instead of word images, utilizing a character-level vocabulary instead of a word-level vocabulary, and training a different parametrization of each of the variants of our very deep learning models. The architectures of these models were otherwise the same as our word-level models.

6. Experiment

6.1. Word-Level Classification Model

We began training our Word-level classification model using VGG-19. Firstly, we found this model to be slow to train given the amount of parameters that it requires. Our initial approach was to use the entire vocabulary, but we found that the run time was too slow. To be able to produce results, we reduced the vocabulary to 50 randomly-selected words that appeared at least 20 times in our dataset. Training was still extremely slow because some of the words that were selected, such as 'the' , had thousands of training examples. We limited each word to 20 examples in order to have sufficient data to benchmark results with a better trade-off on the run time.

Once we narrowed our VGG-19 model, we began training using multiple hyperparameters. Our first runs produced poor final training and validation accuracy because our learning rate was too high. We raised the learning rate of our Adam Optimizer and got 28 percent for training accuracy and 22 percent for validation accuracy.

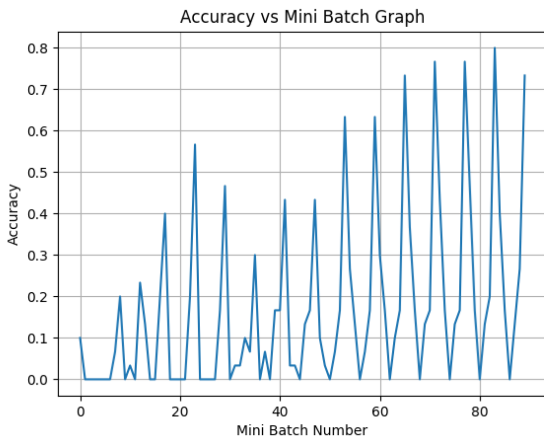


Figure 4. We initially set our learning rate too high when training, which lead to a fluctuation of accuracy.

We realized that VGG-19 was quite slow as a result of the amount of parameters it uses, so we sought other architectures to compare to. The CNN-benchmarks we found [13] hinted that RESNET-18 could produce similar results at much higher speeds. Furthermore, we also believed that the residual layer of the RESNET architecture that can be learned by our model would help our accuracies especially with deeper networks since the residual layer would be adjusted/learned to achieve optimality by our model.

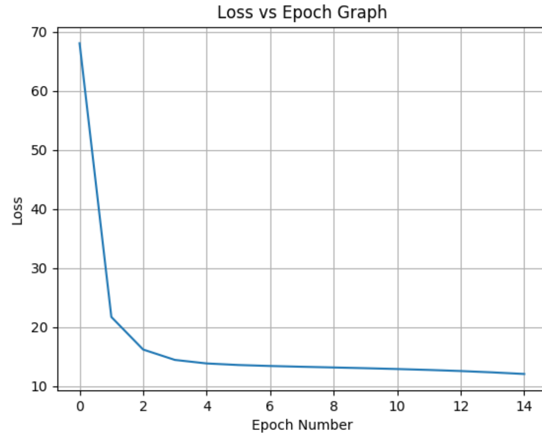


Figure 5. After making revisions to our learning rate and model input, we found that we found we had a consistently decreasing loss.

One technique we have found to be useful during experimentation was keeping the number of epochs low when trying out different hyperparameters. Since our training and validation processes took a long time, and we wanted to find the optimal hyperparameters, we trained our model on 2 epochs, compared the results, and reran our model on more epochs with a subset of hyperparameters that achieved the best results and the ones whose loss/accuracy graphs looked the most promising. This approach certainly saved us a lot of time in training; however, it also had its own disadvantages. For instance, there could have been some hyperparameters that would have taken a longer time to converge, but would have reached optimality in the long run. We might have missed those hyperparameters if their graphs did not look promising enough and their results were poor when compared to the other hyperparameters. This was a challenge we faced due to scarcity of resources, but we tried to eliminate it by considering the graphs of the hyperparameters in addition to the raw numbers that they produced.

At the beginning of our training efforts, the validation accuracy was effectively random and this was because the weight updates were dominated by one class only, and at the end, all the examples were classified as that class. After having printed the weights and their updates for two examples, we realized that the issue was indeed with the weights and the updates were too large, saturating weight values at the beginning and preventing learning.

Given our results of RESNET-18, we deepened the net-

Figure 6. Word Level and Character Level Classification Results

Architecture	Training Accuracy	Validation Accuracy	Test Accuracy
VGG-19	28%	22%	20%
RESNET-18	31%	23%	22%
RESNET-34	35%	27%	25%
Char-Level	38%	33%	31%

work by implementing RESNET-34 to use with our model. We achieved a training accuracy of 35 percent and validation of accuracy of 27 percent.

We chose a relatively large mini-batch size of 50 because if we picked a smaller mini-batch size, parameter updates would happen very frequently, slowing down our model. A larger mini-batch size would capture too much information at once to make a representative step in the direction of the gradient. This was important for our model because we needed to have semi-frequent updates while also not having a corrupted value for the gradient.

Qualitatively, the results of word classification are promising; they show we can achieve decent accuracy for a very complex problem, even given data limitations. However, we realize that higher vocabulary sizes will make the data necessary hard to obtain and also reduce accuracy significantly, which would undermine the ability to accurately translate handwritten text to a digital form. This led us to begin exploring the character segmentation approach.

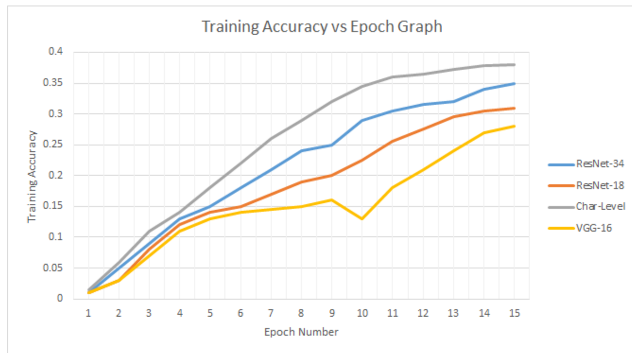


Figure 7. A graph of our training accuracies for each epoch with different architectures for word segmentation and for character segmentation

6.2. Character Segmentation Model

In order to improve the results of directly doing classification for a word, we moved on to segmenting characters and then reconstructing the word by classifying each character individually. First, we downloaded the Tesseract LSTM with convolution model pretrained on the aforementioned English language dataset and then finetuned the model on our dataset [14]. We had to transform our input data to include not only the input image but the bounding box information for each of the component characters (that is, the labels of the characters and the four-corners–best approximation of the top leftmost, bottom leftmost, top rightmost, and bottom leftmost positions of each character), which we extracted from the XML data.

Tesseract contained scripts to generate this dataset automatically given the relevant information [14] We then made a character vocabulary consisting of uppercase and lowercase letters and single-digit numbers. Our final model ended up also using the same optimization procedure, Adam, but a different form of loss suited for the problem called CTC loss. Briefly, CTC (Connectionist Temporal Classification) is a method/loss function developed by Graves et. al. for training recurrent neural networks to label output sequences (labels for characters in our case) from unsegmented input data (the input word image also in our case)[1]. Also used for such tasks as labeling speech signal data with word-level transcriptions, RNNs with CTC loss have been demonstrated to be more effective than more traditional approaches such as CRFs (Conditional Random Fields) and HMMs because of their automated learning process, only needing an input/output data representation as opposed to large hand-engineered features, and can more generally capture context over time and space in their hidden state. Roughly, CTC can segment and label unstructured input data by evaluating network outputs as ”a probability distribution over all possible label sequences, conditioned on a given input sequence” created by considering segmentation of the input sequence given a certain minimum segment size and starting training of the network from there [15]. As stated before, after finetuning and finalizing our segmentation model, we passed in these final segmented character images to our model.

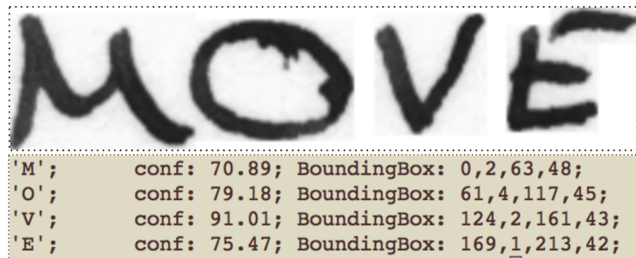


Figure 8. Using the Tesseract LSTM, for every word, we can segment out each character. For instance, when feeding in the word ”MOVE”, we retrieve the coordinates of where each character begins and ends.

As demonstrated in the accuracy graph for character v. word-level classification, we found that the character-level classification was more successful given the same model architectures for classification and even given the potential for error in the segmentation. These results confirmed our intuition that the much smaller scope of the model's initial feature representation problem for characters as opposed to words and final labeling problem helped boost the performance. We believed that our model did not perform even better because of the lack of sufficient data for the scope of our problem and imperfections in the segmentation model. Our model, as most segmentation models do, struggled with segmenting cursive characters because of the breakdown in boundaries in between some cursive characters. In fact, since we trained our segmentation model separately from our character classification model, we were not able to finely attribute the cause of error to our classification or segmentation model. Lastly, there is great diversity in handwriting for particular words/characters among writers, thus making the task of recognizing all of the different ways in which a character or word is written very challenging. We believed that, as often is the case with deep learning problems, even more data (over the course of millions more words) would have helped our model learn a more generalized feature representation for the problem to help it perform better.

7. Future Work

Firstly, to have more compelling and robust training, we could apply additional preprocessing techniques such as jittering. We could also divide each pixel by its corresponding standard deviation to normalize the data.

Next, given time and budget constraints, we were limited to 20 training examples for each given word in order to efficiently evaluate and revise our model.

Another method of improving our character segmentation model would be to move beyond a greedy search for the most likely solution. We would approach this by considering a more exhaustive but still efficient decoding algorithm such as beam search. We can use a character/word-based language-based model to add a penalty/benefit score to each of the possible final beam search candidate paths, along with their combined individual softmax probabilities, representing the probability of the sequence of characters/words. If the language model indicates perhaps the most likely candidate word according to the softmax layer and beam search is very unlikely given the context so far as opposed to some other likely candidate words, then our model can correct itself accordingly.

References

[1] Alex Graves, Santiago Fernndez, Faustino Gomez, Jrgen Schmidhuber, Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks, Proceedings of the 23rd international conference on Machine learning. 2006

[2] Convolutional Neural Network Benchmarks: https://github.com/jcjohnson/cnn-benchmarks

[3] Elie Krevat, Elliot Cuzzillo. Improving Off-line Handwritten Character Recognition with Hidden Markov Models. [4] Fabian Tschopp. Efficient Convolutional Neural Networks for Pixelwise Classification on Heterogeneous Hardware Systems [5] George Nagy. Document processing applications.

[6] Mail encoding and processing system patent: https://www.google.com/patents/US5420403

[7] Kurzweil Computer Products. http://www.kurzweiltech.com/kcp.html

[8] H. Bunke1, M. Roth1, E.G. Schukat-Talamazzini. Offline Cursive Handwriting Recognition using Hidden Markov Models.

[9] K. Simonyan, A. Zisserman Very Deep Convolutional Networks for Large-Scale Image Recognition arXiv technical report, 2014

[10] Lisa Yan. Recognizing Handwritten Characters. CS 231N Final Research Paper.

[11] Oivind Due Trier, Anil K. Jain, Torfinn Taxt. Feature Extraction Methods for Character Recognition–A Survey. Pattern Recognition. 1996

[12] Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. Neural Information Processing Systems (NIPS), 2015

[13] Tesseract Model: https://github.com/tesseract-ocr/tesseract/wiki/TrainingTesseract-4.00

[14] Tesseract 4.0: https://github.com/tesseract-ocr/tesseract/wiki/4.0-with-LSTM

[15] How many words are there in the English language?: https://en.oxforddictionaries.com/explore/how-many-words-are-there-in-the-english-language

[16] Thodore Bluche, Jrme Louradour, Ronaldo Messina. Scan, Attend and Read: End-to-End Handwritten Paragraph Recognition with MDLSTM Attention.

[17] T. Wang, D. Wu, A. Coates, A. Ng. "End-to-End Text Recognition with Convolutional Neural Networks" ICPR 2012.

[18] U. Marti and H. Bunke. The IAM-database: An English Sentence Database for Off-line Handwriting Recognition. Int. Journal on Document Analysis and Recognition, Volume 5, pages 39 - 46, 2002.

[19] Y. LeCun, L. Bottou, Y. Bengio and P. Haffner: Gradient-Based Learning Applied to Document Recognition, Intelligent Signal Processing, 306-351, IEEE Press, 2001

[20] Zhang, X., He, K., Ren, S., Sun, J.: Deep residual learning for image recognition. In: CVPR. (2016)