Handwriting Recognition using TensorFlow

Isaac Anderson, Isaac Gasparri, Riley Van Heukelum,

and Eric Waterhouse

Department of Physics, Computer Science and Engineering, Houghton University,

Houghton, New York, USA

1) Introduction

Machine learning is an invaluable tool with great potential to help better society. Already, this technology is being used in a variety of fields to make people's lives easier. One area where machine learning is of great use is recognizing handwriting in images and converting it into digital text. In an increasingly digital age, the ability to quickly and efficiently transfer handwritten documents to usable computer files is incredibly important.

For example, having digital copies of documents online can greatly improve accessibility for the visually impaired with the ability to increase font sizes, switch to color-blind friendly color palettes, and utilize text to speech tools. Education is another area where these types of machine learning tools could be heavily utilized in the future by allowing teachers and students to convert their handwritten notes from in class to computer documents for later use when reviewing material. Furthermore, the ability to decipher and upload historical documents to the internet without the somewhat intensive labor of manually copying the texts increases accessibility to the knowledge they contain for researchers and other interested parties. Finally, an increased emphasis and use of digital documents also makes it much more difficult for files to be lost or destroyed and helps to save resources by preventing paper waste.

Despite how simple the task of reading and converting handwritten text may seem for a human, it's actually a quite complicated task for a computer. The vast variability between different handwriting styles makes it difficult for a computer to learn specific patterns to correlate with different characters. Handwritten characters can greatly differ in size, shape, orientation, style and more, so much so that no two people have the same handwriting style. As a result, training a machine learning model to do this task well requires a massive amount of data so that the computer can learn from a wide range of examples while trying to generalize the defining features of each letter.

For our project we used a dataset of handwritten names and used a Convolutional Neural Network (CNN) to try and create a model that would recognize the text in the images and be able to output the name as digital text. Converting names, while potentially more challenging than converting normal words due to names often having abnormal spellings, is especially useful in areas such as the ongoing research in natural language processing, as well as healthcare, where having your correct information is extremely important. Despite the complexity of the task of

creating a handwriting recognition model, we decided that it would make for an interesting project and push the boundaries of our knowledge and skills through which we could grow and learn more about machine learning as a whole.

2) Problem Formulation

The task of handwriting recognition and conversion is a sequence-to-sequence (Seq2Seq) task. Seq2Seq tasks are often used in natural language processing (NLP), and work by taking in a sequence of data and reading it token by token, one step at a time, extracting defining features from each step to encode the data into a single, fixed size context vector. This context vector encapsulates the important information from the whole sequence. The encoding step is often accomplished by a Long Short-Term Memory (LSTM) or a Gate Recurrent Unit (GRU) layer. The context vector is then passed to a decoder layer (also often an LSTM or GRU layer) and the output sequence is similarly generated step by step. Each token in the output sequence is created based on the context vector and all of the previously generated output tokens. The process continues until the decoder generates an end-of-sequence token or the maximum predefined length is hit.

In the case of handwriting recognition, the input data for the model is a series of images of handwriting that have been scaled to a consistent size, converted to grayscale, and normalized, represented by arrays of pixel values. A CNN is used to extract spatial features from the images and a Recurrent Neural Network (RNN) is used to process the sequences of features by capturing their temporal dependencies and relationships with one another. The Connectionist Temporal Classification (CTC) function then predicts the probabilities for each character it discovers in the image and outputs a completed sequence of characters that can be compared with the image labels to evaluate the model's performance.

3) Literature Review

A lot of research has already been done in the area of handwritten character identification. A study from 2023 used a CNN to convert handwritten numbers into digital ones, using a dataset of over 70,000 images. In addition, they attempted to see how the model would respond if the digits had extra markings or more rotation to simulate differences in people's handwriting styles. Their model, after several different tests, produced accuracy scores between 85% and 99% respectively, with the biggest struggles coming from those specially edited digits, but performing very well with the majority of the other test images (Jain et al.).

In a study conducted by Google Research, models were tested to see if they could recreate handwritten words using digital ink. Researchers used a few different sized models and trained using images of the handwritten words and the accompanying translation to digital text. The results showed that all of the models had at least 82% of their outputs being classified as reasonable, and the largest model had 67% of results that could pass as human (Mitrevski and Maksai).

A similar research project to ours conducted in 2021 showed that handwritten characters could be accurately identified using a CNN. After building the model and training it on a dataset of images, they found that they had a maximum of 92.91% accuracy on the 200 test images. This study also showed that larger sets of training images led to increases in the model's accuracy. The highest accuracy occurred when there were 1000 images in the training set, but when there was only 200, the accuracy was drastically reduced, to only around 65% (Khandokar et al.).

The fourth study we looked at used a CNN model to convert pages of handwritten sentences to online text. To do this, the model had to be trained to break down the pages into individual characters, methodically starting from lines, to words, to letters. Then, the model, which was trained on over 65,000 images, would identify each character and reconfigure the sentences. After multiple tests, the researchers achieved an average accuracy score of 90.42%, with the main struggles being due to cases of extreme handwriting differences (M, Ghazal T).

4) Method

To achieve this task, we first downloaded the dataset titled Handwriting Recognition, created by user landlord, from the website Kaggle. This dataset contains over 400,000 images of handwritten names and their transcriptions. Each image filename in the dataset was then updated according to our file setup and since the data had been pre-split, we combined the three datasets into a single DataFrame for preprocessing. All null values were dropped, the entries that had labeled "UNREADABLE" were removed, and all labels were converted to uppercase to maintain consistency across the data. We then created and ran a function that filtered the dataset to remove images that had been labeled with unusual characters (characters not in the standard English alphabet). The remaining images were again filtered to remove images with a height less than 20 pixels to remove entries where the name was cut off and not fully visible.

Because of limitations encountered with server resources, the dataset was reduced to 10% of its full size before being split with 70% for a training set, 15% for a validation set, and 15% for a testing set. The images were then further preprocessed by reshaping them to a consistent size of 256 by 64 pixels where larger images were cropped, and smaller images were padded with whitespace. The pixels in each image were also divided by 255 to normalize their values to be between 0 and 1.

The alphabet of expected characters was defined as capital letters A-Z, hyphens and apostrophes. From there, a function was created to encode a given string into numbers representing each character. Variables were created containing the true labels for each image converted to numbers and padded to the max length with -1, the length of each true label without padding, the length of each predicted label, and an output for the loss function. These variables were used to define the CTC loss function.

The model was then constructed using a variety of layers. The first Input layer was used to define the image size as 256 by 64 pixels with 1 channel because the images are grayscale. Then

a convolution (Conv2D) layer was used to extract features from the image with 32 filters and a 3 by 3 kernel size. The BatchNormalization and Activation layers normalized the feature maps and applied a Rectified Linear Unit (ReLU) activation function to introduce non-linearity into the model. After that, a MaxPooling2D layer was used to downsample the data to 128 by 32 pixels while retaining the significant features of each image. A second convolution layer was used to extract further features from the downsampled data with 64 filters and a 3 by 3 kernel size. Again, BatchNormlization and Activation layers were used to normalize the feature maps and introduce more non-linearity. Another MaxPooling2D layer further downsampled the images to 64 by 16 pixels. A Dropout layer was used here to randomly drop 30% of the neurons to prevent overfitting. A Reshape layer converted the data to a tensor with 64 timesteps with 1024 features each and a fully connected Dense layer was used to learn feature representations. From here, a Bidirectional LSTM layer was used to process the image sequences both in the forward and backwards direction which captures temporal dependencies between tokens. Another Dense layer was used to map the output from the Bidirectional LSTM layer to the number of characters in the alphabet defined previously. Lastly, a softmax Activation layer was used to predict the probabilities of each character being recognized at each timestep. With the model created and compiled, it was then trained using the training and validation sets over 50 epochs and with a batch size of 60.

5) Experiments

When working with our model, we quickly learned it was extremely computationally expensive. Though we started working in Google Colab, we quickly reencountered frequent crashes due to using up the available resources. To account for this, we were forced to transition to using the Houghton University servers. We found that our servers had one CPU that gave us a bit more to work with than the 12.7 gigabytes of RAM we had access to through Colab. Unfortunately, the server's GPU was inaccessible, which was reflected heavily in training times. As a result, even when working on the server we encountered strict resource limitations, and it became quite hard to train our model. Even after reducing the dataset to a tenth of its original size and reducing our batch size from 120 to 60 to prevent crashing, it still took over 6 hours to train.

Additionally, we encountered many issues with importing specific packages while on the server. While different Kaggle, and Colab notebooks that we tried were able to use certain useful keras.backend packages, especially the ctc_batch_cost loss function, they refused to work on the server. After some research, this is likely due to these functions being depreciated in newer versions of keras. While we tried downgrading our packages to the appropriate versions, it ended up breaking a lot of other sections of our code. As a result, we were forced to simply reupgrade to the versions we had been using before and find an alternative solution.

When compiling our model, we settled on Connectionist Temporal Classification (CTC) Loss function. The reason we used a CTC loss function was because it allows for a discrepancy between the size of the input and the size of the output and is commonly used for these types of Seq2Seq tasks. This allows our model to account for the wide variability of our inputs and to

process repeated characters. Note for example the name “Jerry” in our database, the repeating ‘r’s would be hard for a naive model to understand, but because of CTC loss we are able to account for this duplication.

We trained our model over 50 epochs, with a batch size of 60 and used an ReLU activation function. Our decisions were heavily impacted by our computationally expensive model. Each epoch took roughly 6 minutes, we were trying to choose a large enough number of epochs to train the model well without overloading the server or further extending the training time. A batch size of 60 was chosen because it worked well with our CNN, Bidirectionsal LSTM architecture allowing for a smoother gradient. Finally, our decision for a ReLU function was due to the fact that sigmoid and tanh functions were simply far too computationally expensive. Rather the ReLU function is a simple conditional check which uses no powers, divisions, or negative numbers which saves neuron space. When initially working with 400,000 images this was a simple decision to make. Another note is that a ReLU adds nonlinearity into our function. The way ReLU takes away nonlinearity is through its “hinge” at x = 0,

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

By ensuring our model was nonlinear we can partially account for how our handwritten data also varies from input to input. For our optimizer we used an Adaptive Moment Estimation (ADAM) optimizer. This was done because an ADAM optimizer combines features of both Momentum and Root Mean Squared Propagation (RMSPROP). The RMSPROP is useful for tracking the exponentially decaying average of our CNN squared gradients and thus adapt the learning rate for each parameter. The Momentum feature in our ADAM optimizer also allowed us to ensure that we could account for the variance in our CNN gradients. The 0.0001 learning rate is used for the ADAM optimizer as per default as it is a good balance between convergence and stability.

After compiling and training the model, we evaluated it on the testing set using two primary metrics. First, we calculated the Character Error Rate (CER) which measures the average number of errors per character in the predicted text compared to the true text. Our model performed abysmally with a CER of 10.063, indicating that there are more errors than characters in the actual label. Looking at the model output, this is likely because the model did not learn to predict blank spaces well and often filled the whole output sequence with characters, regardless of how long the actual name was. We also evaluated our model with the Character Accuracy Rate (CAR) which measures the percentage of characters that were correctly classified. Again, our model’s performance was quite poor with a CAR of 6.77%.

6) Conclusion

Overall, we worked with a large model which was extremely computationally expensive, greatly limiting what we were able to achieve with the resources available to us. Although starting with a larger dataset that would be ideal for training such a complicated model, we were unable to use even close to all 400,000 samples due to limited processing power. From our model's poor performance, we have concluded that the model requires more data to train and a more effective loss function. Our loss function dropped to 137 after our first epoch and did not drop any further. This loss function requires more debugging, but we were simply unable to effectively debug such a large model when a single training session takes 6 hours. Additionally, based on other sources, the ctc_batch_cost loss function from keras.backend would have been much more effective if it didn't cause issues with the server. Based on our current understanding we are unsure fully what is causing our current loss function to behave so poorly but it could be a result of our weights being incorrectly adjusted. This said, we suspect further improvement could be found through more research into the topic.

Since the model took such a long time to train, it was difficult to try and tune any hyperparameters that could make the performance of the model better. While our biggest issue was the ineffectiveness of our loss function at training our model, there might've been underlying issues with our model formulation such as the optimizer, learning rate, and the layers we were using in our model. While the layers we used were based upon research from other similar projects that performed fairly well, they also had access to a better loss function. If there were more time, we might've found that changing some simple hyperparameters in our training and adjusting things such as the number of epochs and the batch size may have improved our model.

We also could've experimented with different methods of building our OCR. Some other ideas included first training a model on a set of handwritten characters and then implementing that into our larger model that works with handwritten names. This could improve our model in that before we even give it a name to train on, it would already know how to recognize individual characters. If we could find a way to split images into different sections based on how much white space was between each character, we could simply make predictions using the character recognition model. We would just need to format the output so that it would output all the characters into a string.

Lastly, we could've experimented with other loss functions. While most articles suggested that when building handwriting recognition models CTC loss is preferred, other articles on CNN models suggested that Cross Entropy loss could also be a viable solution. We could've tried training two models, one with Cross entropy loss and one with CTC loss and compared to see what loss function performed the best. While hyperparameters may have made this issue better, we believe that a different loss function was necessary to make our model train as intended.

Overall, while our model did not perform as intended, we did grow significantly in our understanding of machine learning as a whole and learned a lot about the full process of implementing an algorithm for machine learning tasks.

7) References

Works Cited

Jain, Parth Hasmukh, et al. “Artificially Intelligent Readers: An Adaptive Framework for Original Handwritten Numerical Digits Recognition with OCR Methods.” Information, vol. 14, no. 6, 1 June 2023, p. 305, www.mdpi.com/2078-2489/14/6/305, https://doi.org/10.3390/info14060305.

Khandokar, I, et al. “Handwritten Character Recognition Using Convolutional Neural Network.” Journal of Physics: Conference Series, vol. 1918, no. 4, 1 June 2021, p. 042152, https://doi.org/10.1088/1742-6596/1918/4/042152. Accessed 10 Dec. 2024.

M, Ghazal T. “Convolutional Neural Network Based Intelligent Handwritten Document Recognition - Skyrep.” Skylineuniversity.ac.ae, 2022, https://research.skylineuniversity.ac.ae/id/eprint/158/1/26.pdf. Accessed 10 Dec. 2024.

Mitrevski, Blagoj, and Andrii Maksai. “A Return to Hand-Written Notes by Learning to Read & Write.” Google Research, 28 Oct. 2024, https://research.google/blog/a-return-to-hand-written-notes-by-learning-to-read-write/. Accessed 10 Dec. 2024.

Additional References

Guest Blog. *Introduction to Seq2Seq Models*. Analytics Vidhya , 4 Dec. 2020, https://www.analyticsvidhya.com/blog/2020/08/a-simple-introduction-to-sequence-to-sequence-models/#:~:text=Seq2Seq%20(Sequence%2Dto%2DSequence)%20models%20find%20applications%20in,from%20one%20language%20to%20another

Mwiti, Derrickm. *How to Build CNN in TensorFlow (examples, code, and notebooks)*. Machine Learning Nuggets, https://www.machinelearningnuggets.com/cnn-tensorflow/

Python Lessons, *Handwriting words recognition with TensorFlow*. PyLessons, 23 January, 2023, https://pylessons.com/handwriting-recognition

Rosebrock, Adrian. *OCR Handwriting Recognition with OpenCV, Keras, and TensorFlow*. Pyimagesearch, 24 August, 2020, https://pyimagesearch.com/2020/08/24/ocr-handwriting-recognition-with-opencv-keras-and-tensorflow/

Scheidl, Harald. *Build a Handwritten Text Recognition System Using TensorFlow*. Towards Data Science, 15 June, 2018, https://towardsdatascience.com/build-a-handwritten-text-recognition-system-using-tensorflow-2326a3487cd5

Wedoro, Kebun, and Wibisono, Amelia. *Handwriting Recognition Using CRNN in Keras*. https://www.kaggle.com/code/kebunwedoro/handwriting-recognition-using-crnn-in-keras/notebook#Train-our-model

Wikipedia. *Seq2seq*. Wikipedia, 13 December 2024, https://en.wikipedia.org/wiki/Seq2seq