Handwriting-to-LaTeX Project Report: CS 7643

Nathan Wang
Georgia Institute of Technology
Atlanta, GA
nwang334@gatech.edu

Rohit Tuteja
Georgia Institute of Technology
Atlanta, GA
rtuteja6@gatech.edu

Abstract

Students at the Georgia Institute of Technology have been exposed to courses primarily focused on theoretical and mathematical aspects of the study. For example, homework, reports, and projects in courses such as Algorithms require complex proofs with mathematical backing. In these classes, it is often most convenient to write the thought process, math, and final proofs by hand. While this mostly is not an issue, many can relate to losing some points only to see "hard to read" or similar comments as the grading explanation. From the grading TA's perspective, messy, handwritten work is slower and more difficult to comprehend, which may result in unhappy students or more time required on regrade requests. One solution is to submit work in LaTeX; however many students dread completely rewriting their work in LaTeX, wishing to avoid the process of typing up what they've worked so hard to write down previously. What if this was no longer as difficult a task? What if with just pictures of the handwritten work, it were automatically turned into easy-to-read LaTeX formatted code, without the tedious process of retyping or figuring out LaTeX formatting? This project aims to do just such: take an image of handwritten math, recognize the symbols, and translate into LaTeX-friendly code. Part of the difficulty with this task is dealing with the wide variety and levels of legibility of handwriting, making it an interesting task to handle, and one that may require much more effort than expected.

1. Introduction

1.1. Motivation

As mentioned in our Abstract, the problem we aim to solve is the tedious, difficult translation of handwritten work to a more legible format. Thus, we hope to enable people to write down their work by hand and easily convert it to the LaTeX format. As we will discuss in our Approach Section, we propose to solve this problem by applying multiple deep learning techniques, most particularly a modified encoder–decoder model.

The ideal aim of this experiment is to provide a way for handwritten work to be converted to LaTeX easily, for the benefit of students in Computer Science who may not have the best handwriting when submitting their work, saving them those small points lost and saving TAs hours of frustrating work grading and regrading. If successful, our project will provide an easy conversion from mathematical proofs to LaTeX, which will save time and happiness for students. .

1.2. Background

Best current methods for handwriting to LaTeX applications involve segment by segment writing on an iPad or computer with a stylus. However, we want to create a working solution that can take static images of handwritten work done on paper and convert it into LaTeX post-writeup. Solutions that convert whole documents and layouts to LaTeX, and even just converting mathematical expressions has not been created yet to the best of our knowledge. Regarding the implementation of a solution, we propose a caption generation model that takes in an image input. For our encoder, we explored a variety of PyTorch Convolution Neural Network architectures with a ResNet50 default backbone. This means, we train our CNN from scratch and experimented with different dropouts, hidden layers, activation functions, and embedding sizes. As for our decoder, we explored different LSTM architectures. For example, we tested models with various numbers of LSTM layers, dropouts, hidden layer size, and embeddings.

State of the art solutions for handwriting text recognition and optical character recognition rely on CNN or LSTM models. For example, Tesseract and OCRopus use LSTM-based models to achieve a 1 to 2 percent character error rate on printed books [3]. Papers by Reul et al. and Mishra et al. emphasize the potential of applying CNNs to handwritten recognition tasks. Reul et al. claim that CNNs are best at hierarchical, location invariant data while LSTMs work best for temporal, ordered data. Past work has incorporated convolutions and pooling layers either prior to or after a single LSTM layer with varying results [1]. By incorporating a CNN encoder with an LSTM decoder, we hoped to capture both the temporal dependencies via our LSTM layer, as well as the textual, spatial properties via the CNN. Rather than attempting to run our CNN and LSTM in parallel, we opted to employ the encoder-decoder model mentioned earlier. Our CNN ResNet50 backbone includes 4 convolution layer blocks with region of interest pooling layers and a region proposal network as indicated in the Faster-RCNN paper [2]. As we began implementing this proposed solution, we ran into further design decisions we had to consider.

1.3. Data and Preprocessing

The data used in this project to train and test on was mainly taken from a the CROHME 2019 and 2023 datasets that are full of Handwritten Math Symbols, found at the following link here. This dataset has over 15,000 images with different handwriting styles, providing a large sample of handwritten math to parse. The linked dataset itself covers numbers, mathematical characters, a variety of LaTeX mathematical commands, and LaTeX syntax symbols. We will go more into depth on the nature of dataset balance and usefulness in our section on Tokenizer.

In the project folder, found here, the 'LatexPreprocessing.ipynb' notebook shows some of our pre-processing pipeline for our dataset. The data was formatted into train, test, and validation folders, each having folders for CROHME 2019 and CROHME 2023 data. Within each CROHME folder, data was segmented into IMG and INKML, with each image having the same base filename for easier lookup. First, we saved our data to Google Drive and then iterated through each folder to standardize filenames and match images to their corresponding inkml annotations. Lastly, we extracted the annotation text contenst and saved all images into an IMG folder and all corresponding annotations as txt files stored in the INKML folder. With this pre-processing pipeline, we combined training data into one set for us to extract and use more effectively.

From there, we began data loading and the model pipeline within the 'Latex.ipynb' notebook, which was our first model attempt, and within the 'LatexV2.ipynb' notebook for all future attempts. We used the Dataset and DataLoader classes from the torch packages to take our preprocessed data and create a custom dataset that stores image metadata and corresponding annotation text. The CustomDataset extracts and transforms the images to grayscale when needed. We decided to limit our images to be only the ones of shape (1010x1010) to reduce noise from distorted, different-shaped images upon resizing. Therefore, our preprocessed training dataset has 9,511 datapoints, which is a small dataset to train a deep learning model for our task. Additionally, as we will discuss later in the report, the annotations in this dataset are inconsistent (some annotations have dollar signs for math mode, some annotations have spaces and others do not) and have imbalanced number of tokens.

2. Approach

This section documents our initial approach and will begin with a focus on the ideas and methods that originally were used when starting this project. We will also emphasize critical problems that arose during our model implementation, and explain the impact of each on our the evolution of our approach. Our first, as well as many other attempts (in all we have one CNN-Transformer model, followed by 3 versions of CNN-LSTM models of increasing accuracies), were not successful, so we will discuss our initial attempt in this section with further discussion in the Experiments and Results portion.

2.1. Initial Model Overview

Initially, as evident in our 'Latex.ipynb' notebook, we began with a CNN encoder with a pretrained ResNet50 backbone and a Transformer decoder with multi-head attention. The idea was to use a well-trained CNN model and tune its parameters to fit our LaTeX generation task, as this was an approach we saw work in previous projects. Rather than initially pursue an LSTM decoder as per our proposal, we elected instead to employ a multi-head attention transformer because of our findings from Assignment 4 on Seq2Seq versus Transformers models. For all models, our input is an image, passed into a CNN encoder, which will output into the decoder, which uses these features to predict the next token in the sequence. Our loss is calculated by determining if the prediction matches the target, the same LaTeX string but shifted by one so that the first token of training will be used to predict the next token that represents the first of our target sequence. We use Pytorch modules, Google Colab notebooks, and experimented with backbones, input shape and channel, and other architectural choices.

Our model would not train well, and so we opted to attempt our originally proposed approach: a CNN encoder and LSTM decoder. However, as we will discuss in our Experiments and Results Section, we want to further explore a Transformer decoder with our modified data pipeline and tokenizer.

2.2. Initial Problems

Some of the problems we initially expected were resource dependant, and regarding the ability to train the model efficiently for free on Google Colab with a large data set and the high expected training time due to the nature of this problem (which we knew was extremely difficult and has not been achieved to a notable extent to the best of our knowledge). As a result, we started off limiting the size of our data set to 1600 only, just to try and be able to quickly train models and see if there was any promise before dedicating the time and resources to train on a larger data set.

Another problem we anticipated prior to training our first model was the non-existence of a widely-accepted tokenizer for LaTeX. In our first attempt, we manually created a custrom regex expression to tokenize our latex inputs:

```
\w+|[\[\]{}()=+\-*/_,.]|\\[a-zA-Z]+|\\[^\s]+
```

We decided to split the LaTeX string by the smallest groupings possible while maintaining command names as full tokens. For example, in the regular expression we implemented below, we tokenize if we find matches with specific characters within corresponding brackets, backslashes followed by alphabetic or non-whitespace characters, and sequences of alphanumeric and underscore characters. Although this tokenizer created valid tokens for the majority of input sequences, we later determined through experimentation and analysis that this tokenizer was not optimal, as it created uneven length tokens with some non-sensical choices for splits. This illustrates that our tokenizer misses some splits due to spacing and logical misunderstandings of LaTeX. Our tokenizer created a vocabulary size of 1,759 tokens. We will further discuss the shortcomings of the existing tokenizers for LaTeX in our Experiments and Results section, but even with the TexSoup Python library we employed, non-sensically long tokens became a part of the vocabulary.

As for obstacles faced in later phases of our project, we faced issues with long-running train loops and GPU usage. Training a deep learning model like a transformer or CNN-LSTM encoder-decoder model takes vast resources and standard devices may not suffice. We found that Google Colab's free credits were not sufficient, as we were repeatedly disconnected from the free T4 GPU runtimes. Even before training our first model, our work was computed on a CPU runtime because our Google Colaboratory accounts ran out of GPU credits. Without GPU capability, even creating and loading the dataset took hours longer than it should have. Eventually, we bought Google Colab Pro and set our runtime to use GPU, which significantly increased the speed of our initial tests.

The original model in 'Latex.ipynb' was tested with methods found in 'OldEval.ipynb', and was quickly found to be insufficient, spitting out a one repeated token when trying to predict any captions. This led to a rework of the model, found in 'OldEval.ipynb', reducing the batch size of the smaller data set, still capped to 1600 items for now, with the hope being to provide a more fine tuned model. The core of the model was very similar to before, with much of the changes being in the Decoder. Rather than implementing a transformer, we opted for an easier, more lightweight LSTM decoder, as proposed initially. After achieving some amount of accuracy in LaTeX generation with a CNN-LSTM model, we planned on correcting our CNN-Transformer model to compare the generation accuracy.

We found that when a model was trained, it would still simply spit out repeated predicted captions like "22222222..." or only the 'padding' tokens, again forcing a reconsideration of the correctness of our model. Different modifications were made, such as masking the EOS tokens, ignoring the PAD tokens, correcting our forward pass, and fixing the shifts for special tokens.

We continued experimenting with this CNN-LSTM model and once we began seeing evidence of training, we increased our dataset and number of epochs.

3. Experiments and Results

3.1. Further Experiments

Following our initial failed attempt at a CNN encoder with a transformer decoder, we successfully trained multiple versions of a CNN-LSTM encoder-decoder. For all models, we used a modified version of the Resnet50 backbone for our CNN encoder, followed by a Pytorch LSTM cell or module with varying implementations. As for sequence length for generation, we determined the maximum sequence by taking the maximum length sequence from our training data. For all CNN-LSTM models, we utilized our full 9,511 datapoint dataset, and other than our last version, we resized the (1010x1010) images to (128x128).

Our process to create our CNN-LSTM model was filled with numerous issues and debugging print statements to validate the correctness of tokenized sequences, byte-converted images, special token shifts for training, padding handling for batched sequences, proper generation scripts, and much more.

Once we managed to successfully train and generate LaTeX for our first CNN-LSTM model, we analyzed losses and test generations to determine which hyperparameters, model architecture, and other changes to incorporate. By the end of our project, we created three versions of our CNN-LSTM model with varying success.

3.1.1 CNN-LSTM V1 Model

Our first version of the CNN-LSTM model utilized our regular expression tokenizer with a vocabulary size of 1,759 words. For our encoder, we used a pretrained Resnet50 backbone with 2,048 hidden layer neurons in the fully connected layers and a ReLU activation function. Our LSTM decoder employed a hidden layer size of 1024 (which we later realized was too large) and calls the LSTM cell on the extracted image features from the encoder followed by subsequent calls on the caption embeddings. In all models, we specify the embedding size as well, which along with the vocabulary size, is required for the encoder and decoder. In this model, we used 181 as the embed size.

In terms of hyperparameters, we chose a learning rate of 0.0005, trained for 2 epochs, and used the Adam optimizer.

We initially thought that this approach would primarily factor in the image as the main input and use the caption ground truths as a teacher forcing technique. However, as we analyzed the training loss and experimented with generating LaTeX, we found that this model learned to output the exact same sequence no matter the input image. This model's training loss was low but generated gibberish, indicating that our model was not properly using the features from the CNN encoder and rather learning to copy captions.

3.1.2 CNN-LSTM V2 Model

Our architectural changes for this version lied primarily in the fixes mentioned below and parameter choices. For our encoder, we decided to use an untrained Resnet50 backbone with 4096 hidden layer neurons and a ReLU activation function. Our LSTM decoder used a hidden layer size of 512 and an embedding size of 512.

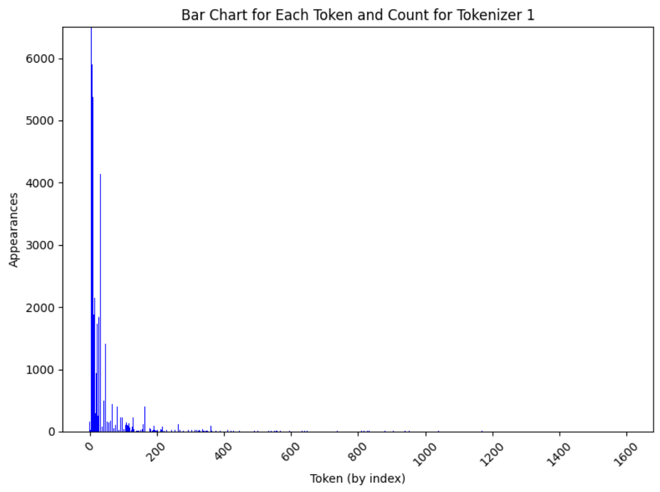
Our hyperparamters were a learning rate of 0.00005, trained for 10 epochs, and with the Adam optimizer with weight decay (L2 regularization) in case of overfitting.

Our second iteration of the CNN-LSTM model started to generate different sequences for images. However, we found that the model seemed to overfit the data due to token imbalances in our training data. We determined this issue from our observations of repeated use of "mbox" and "frac" in output generations when the image had no such commands. To understand the reasoning behind this observation, we printed and plotted the distribution of tokens for a sample of the dataset.

We found that tokens like curly braces, "frac", "mbox", and approximately 20 of our tokenizer's 1,759 tokens had massively more appearances in our dataset compared to the rest. Thus, we pursued other tokenizers and found the TexSoup Python library that provided us with a vocabulary size of 5,759 and a slightly better distribution of token appearances. In order to mitigate this imbalance more, we elected to incorporated manually chosen weighted cross entropy loss, which we stopped using in future model versions. In Figure 1a and Figure 1b, we can see the distribution of tokens by appearance in our dataset. Although TexSoup finds 5,759 tokens while our tokenizer finds 1,759, the distribution of tokens is roughly the same (with TexSoup slightly better).

At this point, we experimented with our training loop, printing out predicted responses to caption targets, and we realized we forgot to account for PAD tokens, EOS masking, and improperly shifted the targets with respect to the inputs. In order to fix these mistakes, we consulted example

[a]



[b]

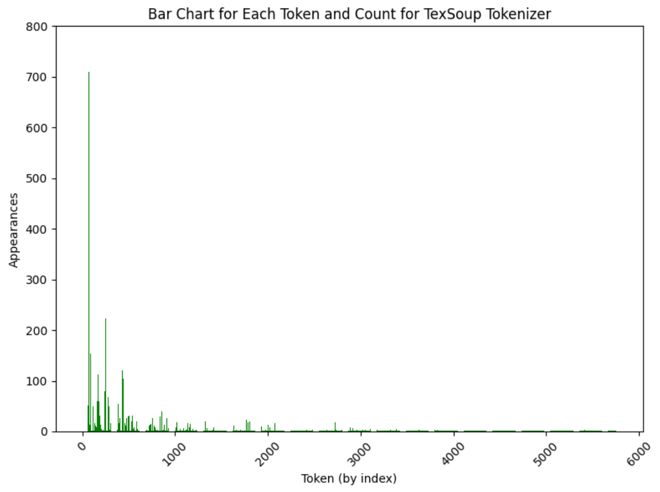
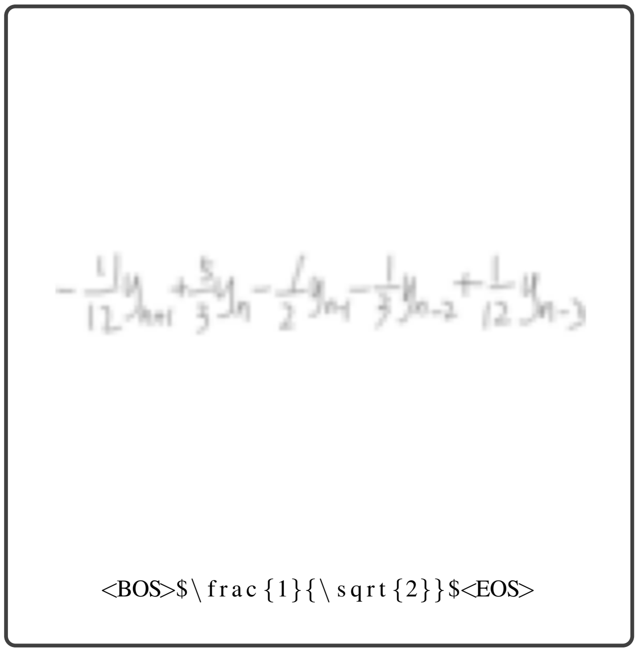


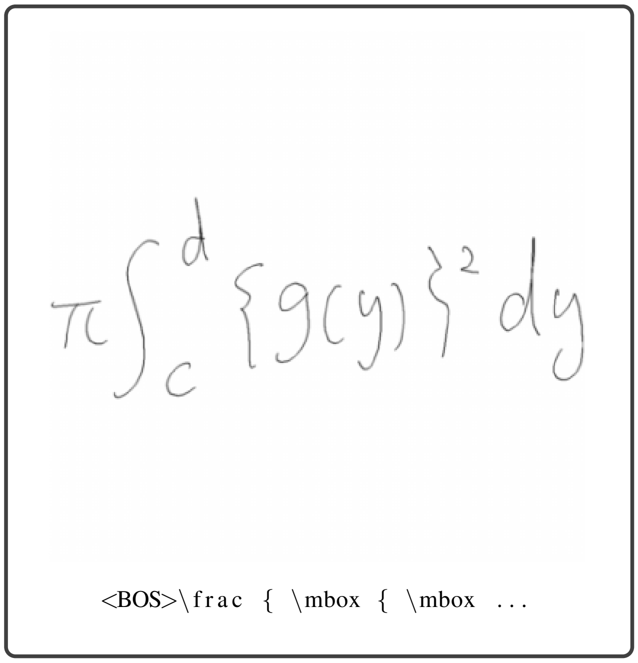
Figure 1. Comparison of our tokenizer and TexSoup tokenizer. TexSoup does not have as drastically skewed tokens as shown by the distribution.

code on Github [5] and Medium [4] to better understand the purpose and implementation of certain LSTM features with Pytorch. Once we fixed these mistakes, we gradually saw our model start to predict better LaTeX. For example, our model would begin predicting gibberish or all curly braces (due to the token imbalance), but with more iterations we began to see EOS at the correct positions, curly braces starting to match up in sequential order, and commands or alphanumeric characters starting to fill into the holes. This seemed extremely promising, and when we tested the generation on images, we found that on "easier to read" and less complicated images, our model would generate decently. Although never outputting the correct response, one of our better predictions generated the following:



```
<BOS>$\frac{1}{\sqrt{2}}$<EOS>
```

Figure 2. Example of input image and generation for CNN-LSTM V2 model. Note that this image is 128x128 and the result is blurry.



```
<BOS>\frac { \mbox { \mbox ...
```

Figure 3. Example of input image and generation for our 1 layer CNN-LSTM V3 model. The mbox command is generated until the end.

3.1.3 CNN-LSTM V3 Model

Architecturally, we experimented with different LSTM decoders. More specifically, we tested training with increased hidden size and embeddings 1 layer LSTMs, 2 layer LSTMs, 1 layer bidirectional LSTMs, and added Dropout and Swish activation functions. 2 layer and bidirectional LSTMs took a long time to train, and we decided to use 1 layer as our Version 3 model. Our embedding size was 320 and the hidden layer size was 512 with a (404x404) image for less blurry images. As seen in Figure 2, the input image of size (128x128) results in a too blurry image. We thought that if we kept a less blurry image, the model could learn better. We also converted the image to grayscale, since handwriting does not require RGB and we can use the excess memory elsewhere. However, we found that using a two layer LSTM resulted in too many parameters and largely overfit our data while our one layer LSTM underfit our data.

In terms of hyperparameters, we chose to use a learning rate of 0.00003, trained again for 10 epochs, and kept weight decay.

This version of our model seems to overfit our data and results in worse generation than Version 2.

CNN-LSTM V4 Model For this version of our model, we focused on reducing overfitting. We reduced CNN hidden dimension from 4096 to 2000, chose an embed size of 164, as LSTM decoder with hidden layer size of 450, weight decay of 1e-5, and learning rate of 0.0005. We stuck with the same (404x404) image size and re-employed weighted cross entropy loss but with higher weights.

We found that for images that our model is less confident predicting, it tends to predict the most likely of a set most seen sequences: if the image looks like it includes frac, limits, or mbox, it will generate a set sequence of tokens that are the same across other similar predictions. Nevertheless, this model predicts different sequences for different images most of the time, which is a good sign on reducing overfitting. On short, simple LaTeX images, our model works decently compared to previous versions, but as the model generates more tokens, it starts to lose contexutalization. Still, the model tends to stick the same sequences of tokens together for certain image characteristics, and seems to be beginning to memorize the most likely outputs rather than considering the image. Thus, in future iterations of this project, we need to increase and balance our dataset and create better tokens. We may also want to consider employing multi-head attention.

3.2. Evaluation Metrics

The evaluation metrics we focused on for our generation was human expectations, training loss, and other debug strategies we employed to achieve what we have. In order to test, we created a generation script that passes in an image that is encoded using our CNN encoder and then is passed into our decoder along with a BOS token. As we generate until max sequence length or EOS early stopping,

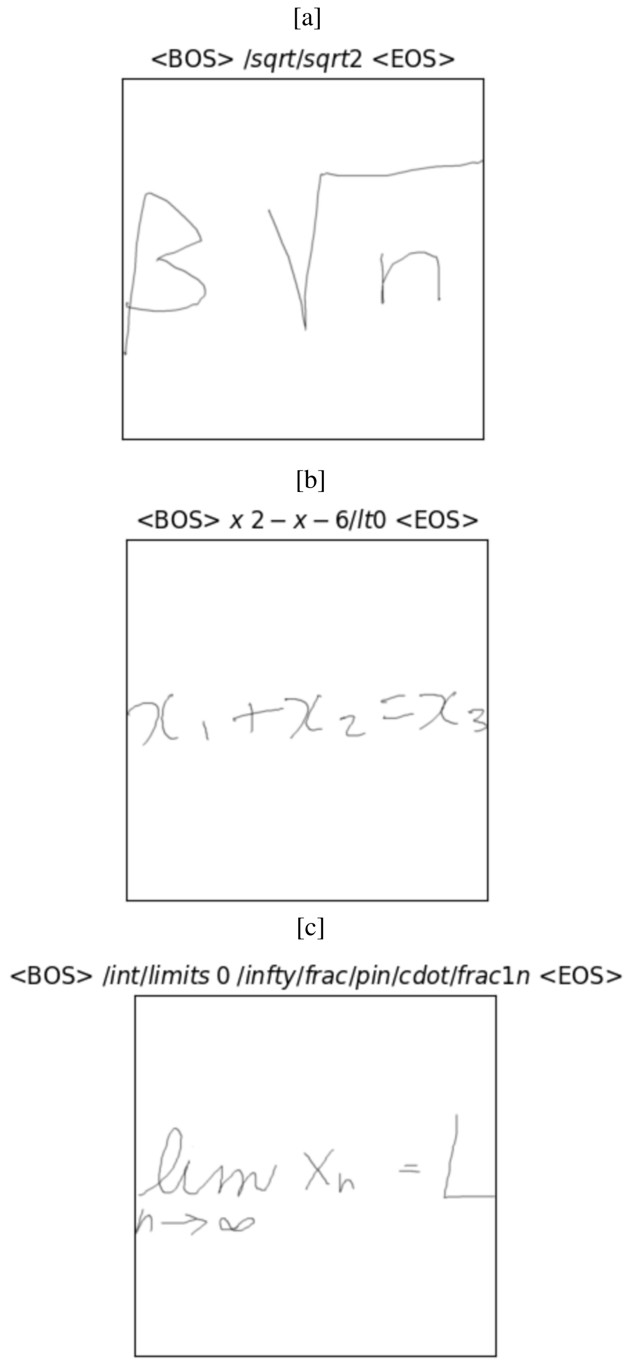

[a]
<BOS> /sqrt/sqrt2 <EOS>
[b]
<BOS> x 2 – x – 6//t0 <EOS>
[c]
<BOS> /int/limits 0 /infty/frac/pin/cdot/frac1n <EOS>

Figure 4. Comparison of generations with a longer versus shorter LaTeX image input with our CNN-LSTM V4 Model.

we append the next token to the input sequence. We test for a random sampling of 10 images. Overall, our models did not perform that well due to multiple possible reasons: the small, limited dataset with inconsistent annotations; the complexity of tokenizing LaTeX code; the time and memory constraints when attempting to train larger models; and others.

As shown in the figures, our models did not predict too well, which would result in near-zero metric scores across various metrics. Despite this, we incorporated ROUGE scoring into our evaluations simply to see if there were any cases in which our prediction had notable similarities, only to find that in most cases our scores were all 0 - in some cases, however, they were slightly above 0, but for all intents and purposes, the scores confirmed to us that our model was not accurate in any meaningful way; though, it was trying as evidenced by the small similarities in some cases, shown by those few non-zero values.

The limitations of the dataset is explained in our Data and Preprocessing section. As for tokenizer, we could instead find or create a tokenizer that groups commands together. For example, rather than splitting frac, {, }, and digits, we could instead treat the entire frac{1}{2} as a token. This could mitigate our issue of imbalanced number of tokens but will prove much harder to tokenize.

3.3. Results Evaluation

Quantitatively, our V1 CNN-LSTM model achieved a loss of roughly 0.63, which did not reflect how this model did not learn based on the image at all and would generate the exact same gibberish. Versions 2 and 3 achieved losses of roughly 1.8 and 0.6038 respectively. Both models generated different sequences for different images, but Version 3 seemed to overfit, evident by the fact that about 8 of the 10 tested images output similar or same incorrect generations (Version 2 saw 6 out 10). Our best model (Version 4) saw similar or same generations about 4 out of 10 times and reached a training loss of 0.9395.

Qualitatively, all models did not predict the correct sequence of LaTeX, but the versions 2 and later were often able to generate the correct length of the sequence, locations of curly braces and frac commands, and understand where commands should go relative to symbols and braces. The fact that the models learned some commands and symbols better may be due to the imbalance of tokens, as applying weighted cross entropy as in versions 2 and 4 helped slightly. As shown in Figure 4, our Version 4 model somewhat learns some tokens and for the tokens later down the line or that it may not know, it generates the most seen, most likely token like frac or sqrt.

Overall, we would say good progress was made in understanding and attempting to translate handwritten images to LaTeX, but limitations like datasets, tokenizers, and hardware/system constraints prevented better results (mostly due to overfitting of our model). In future attempts, we hope to get a working multi-head attention transformer decoder working, a larger dataset, and a good tokenizer.

4. Conclusion

Our goal for this project was to create a deep learning model that can take image inputs of handwriting and convert it to LaTeX for easy translation to a legible format. We initially implemented a CNN encoder with a multi-head attention transformer decoder but later opted for a various attempts of a more simpler CNN encoder with an LSTM decoder.

A majority of our time was spent debugging models and verifying that the training and generation loops work as we are expecting. After achieving a working base model, we experimented with architectural, hyperparameter, and tokenizer changes. From our analysis, we found that the best embedding size for our task lies between 320 and 512, the optimal LSTM hidden layer size is less than 1024, the CNN hidden layer size can be around 4096, and the 1 layer LSTM model works best across our human testing metrics and training loss. We also determined that both our tokenizer and the open source Python library LaTeX tokenizer need to be improved upon or a cleaner annotated dataset should be selected.

Throughout the project, we had to consider common deep learning pitfalls like overfitting versus underfitting, hyperparameter tuning, data preprocessing, LSTM model choices (like embeddings), and more. That being said, the lack of computational power, or more so the lack of reliable access to such, hindered performance a lot as well, slowing us down a lot in the beginning of the process. Much was learned about the scale of ”deep learning” projects and the resources and complexity that go into them, and while the final model may not work as well as hoped, the experience serves as a good stepping stone to bigger, better projects in the future.

5. Contribution Table

As requested, the contribution table can be found in Table 1, breaking down who contributed to what over the course of this project.
All of our data, preprocessing scripts, models, configs, and other code can be found on Google Drive at **this link**.

6. References

[1] Mishra, Atman, and A. Sharath Ram. ”Handwritten Text Recognition Using Convolutional Neural Network.” arXiv preprint arXiv:2307.05396 (2023).

[2] Ren, Shaoqing, et al. ”Faster r-cnn: Towards real-time object detection with region proposal networks.” Advances in neural information processing systems 28 (2015).

[3] Reul, Christian, et al. ”Improving OCR accuracy on early printed books by utilizing cross fold training and voting.” 2018 13th IAPR International Workshop on Document Analysis Systems (DAS). IEEE, 2018.

[4] S. Ulyanin, “Captioning images with pytorch,” Medium, https://medium.com/@stepanulyanin/captioning-images-with-pytorch-bc592e5fd1a3.

[5] Tatwan, “Tatwan/image-captioning-pytorch: Image captioning using CNN+RNN encoder-decoder architecture in pytorch,” GitHub, https://github.com/tatwan/image-captioning-pytorch.

Student Name	Contributed Aspects	Details
Nathan Wang	Data Processing, Model Implementation	Processed raw data set and built data set for the project, implemented the architecture of the models, such as the Encoder/Decoder for the CNN and LSTM layer. Built most of the different versions of the model, iterating architecture each time, and wrote most of the evaluation scripts.
Rohit Tuteja	Training, Tuning, Bug Fixing	Trained the model, fixed many of the run time errors that came up and made sure things like all the dimensions lined up properly. Caught any mistakes that may have been made, acting as quality control in a sense. Tuned the model through its hyperparameters between runs to improve results. Implemented the ROUGE score metrics and helped analyze results when deciding on how to improve model.
Equal Contributions	Decisions on Project/Model, Report	Both members equally contributed to choosing the project, model type, writing the report, and getting Colab credits to train the model, as well as making decisions regarding model architecture.

Table 1. Contributions of team members.