Brian Zhu
Tejasvi Kothapalli
May 10th, 2021

# CS 194-80 Final Project Report

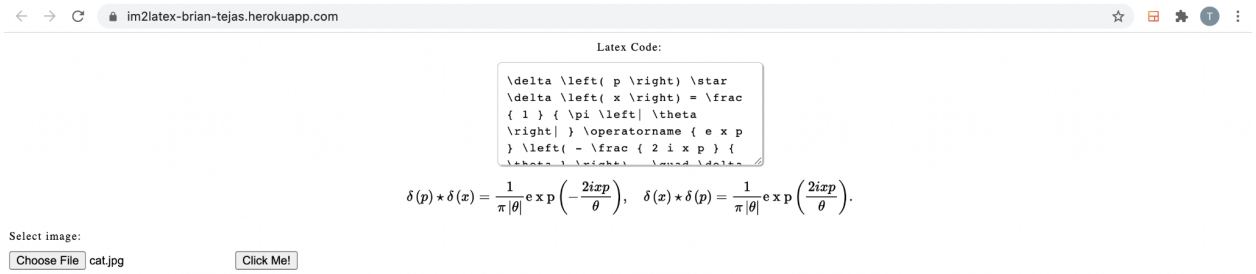
For this project, we created a model that converts images of LaTeX into its original expression. The project is based on previous research done on the exact same task. Since the paper was from more than 5 years ago, we wanted to update the model and see if it is possible to make it perform well and hopefully exceed the results from the paper. Our next step after that is to deploy this model in an application that provides a sandbox for rendering LaTeX expressions, while also having an OCR feature for converting images to text.

## Final Model Architecture

| Encoder: |
| --- |
| ResNet18 (modified where 7x7 has 1x1 stride, removed average-pool and FC layers)<br>1x1 Conv2d, 512 in 512 out (512 out is transformer dimension)<br>Positional Encoding in 2D space (paper, source code) |
| Decoder: |
| Embedding<br>Positional Encoding in 1D space (source code)<br>TransformerDecoder (512 dim, 4 heads, 4 layers)<br>FC layer |

## Deployment

The lackluster UI of the webapp does not do justice to the effort required to deploy the model.



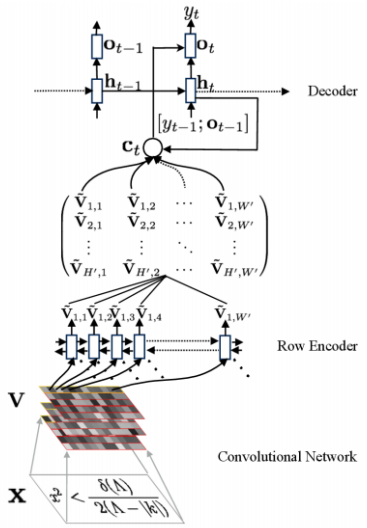

The goal of the UI is to allow users to upload an image containing latex and have the markup uploaded to the text area as seen above. It takes around five seconds for the request to run (gpu was not used on server). The deployment process itself was done by creating a python flask server and hosting it on heroku. The frontend sends image data to the flask server, which runs it

through the model. The backend then sends back markup to the frontend. The frontend then renders the latex markup code. While the webapp was ultimately hosted on Heroku, we also wrestled with Amazon AWS, Google Cloud, and Render. We found that Heroku has the easiest deployment process for beginners!

*Web App Link*: https://im2latex-brian-tejas.herokuapp.com/
*Curl Request*: curl -X POST -H "Content-Type: multipart/form-data" https://im2latex-brian-tejas.herokuapp.com/predict -F "image=@<PATH TO IMAGE>"
*Github Link*: https://github.com/tejasvikothapalli/latex2

**Deciding on the Model Architecture**

One of the core challenges we faced during this project was finding a model architecture that would perform well.




In the original paper, Deng et. al. describes their Image-to-Latex model as a CNN encoder followed by an LSTM neural machine translation model. More concretely, an image is passed through a CNN to create a feature map. Each row of the feature map is then fed into an LSTM encoder. The features produced by the row encoder are used by the LSTM decoder in an attention mechanism to generate the LaTeX text.

Since this paper was written in 2015, our first iteration replaced each component of the original model with something more modern. The CNN is a ResNet18 up to the average pool, LSTM encoder is now a transformer encoder, and the LSTM decoder is now a transformer decoder. The paper did not talk much about how the feature map was set up, so our first problem was to figure out how to shape the ResNet output such that it can be utilized by the transformers.

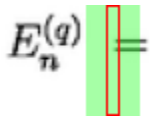
In a vanilla ResNet, when the image reaches the average pool, its dimensions have been shrunk by 32 times due to the 5 2x2 stride operations throughout the ResNet. The resolution of the inputs ranged from 40 x 160 x 3 to 100 x 500 x 3, so by the time they reach the average pool,

the resolutions are about 1 x 5 x 512 to 3 x 15 x 512. Considering how the maximum sequence length of the data needs to be 150 tokens, we were concerned that having a maximum of 45 pixels (and thus 45 512-dimensional vectors) in the output will not yield enough information to be used by the transformers. We also were concerned that the wide range of image resolutions created too much variation in the amount of information being propagated to the transformers.

To combat these two issues, we first changed the strides in ResNet to all be 1x1, so the image retains its shape. Then, we fix the output shape of the ResNet to be 1 x 150 using the average pool. Since the image sizes were wider than 150 pixels, there was not much concern in average pooling to a 1 x 150 x 512 volume. Finally, to support a custom transformer dimension (which we left at 512), we placed a 1x1 convolution layer before the average pool such that the output channel matches the transformer dimension. The model was trained using Adam with a learning rate of 3e-4 that decays by 0.1 every 2 epochs.

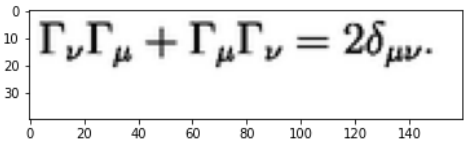
The model did not perform well at all. During this first iteration, not much of the loss or accuracy was recorded, but when running predictions with the model, the output only consisted of left and right braces, which clearly indicated poor performance. We suspect that the average pooling was the source of poor performance for two reasons.

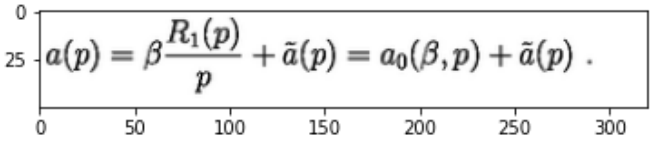
The first reason is that the average pool does not cover the parts of an image in an informative way. Take the figure below as an example:




The average pool splits the feature map into 150 equally sized vertical columns. The red box would represent the area a particular output pixel covers in the input feature map, while the green box represents the receptive field of that same pixel on the original image. We see that the receptive field is majority whitespace, so the final pixel would mainly average together information that encodes whitespace. Since the pixel is mainly information about whitespace, it would not be implausible for the model to think that this area is a brace. Braces (not "\{") can be thought of as "boundaries" in the expression and have no direct mapping to a symbol, and it could be possible that the model learned to utilize it as a placeholder for whitespace. Since many of these average pool pixels would encounter similar behavior, it would make sense that the model outputted a lot of braces. Furthermore, the average pool is unable to capture any information about vertical structure (e.g. from a fraction or summation).

The second reason average pooling may have led to poor performance was the reason we used it: to fix the output dimensions of the ResNet. Side by side are a 4 x 160 (left) and a 50 x 320 image (right):



We see that larger images would tend to have more tokens in the output. Without an average pool, the number of pixels in the output of ResNet would be proportional to the resolution of the original image. The amount pixels itself can help encode information about how many tokens there are in the latex expression. By using an average pool layer, we are fixing the number of output pixels and thus throwing out what potentially could be valuable information.

Thus, in our second iteration of the model, we removed the average pooling layer. To control the number of output pixels, we readjusted the strides in ResNet18 such that only the 7x7 convolution at the beginning has a 1x1 stride, while the remaining blocks were reverted back to the original 2x2 strides. This means that the ResNet output will have dimensions that are 16 times smaller than the original image, ranging from 3 x 10 x 512 to 7 x 30 x512, or from 30 to 210 512-dimensional vectors. This was a reasonable amount of vectors to feed into the transformer encoder.

Using this architecture, the model was able to perform better, reaching about 55% validation accuracy, where accuracy was defined to be the number of correct words in the output compared to the target. The words were chosen using greedy search on the logits. Looking at the text that was predicted, the model was beginning to output some symbols, but it was unable to grasp the general structure of the LaTeX in the image.

Finally, we decided to try removing the transformer encoder in our third iteration of the model (effectively creating the ResNetTransformer used in the labs). Surprisingly, the model performed significantly better, reaching 95% accuracy. The text predicted by the model matched the structure of the text in the image, and occasionally mislabelled some symbols. The source of most error was from decorators like “\hat”, “\dot”, “\tilde”, etc., and from mislabelling symbols in sub and superscripts.

It is hard to come up with a theoretical explanation for why removing the transformer encoder yielded such improvements, but we believe it is an example of Occam’s razor. In the original paper, the CNN encoder that was used may not have been very expressive, so the authors used an LSTM to help encode information more effectively. Now, ResNet may have become such a good feature extractor where appending a transformer encoder is like adding deadweight to the model. The transformer encoder would add little to no benefit to performance, while making the model significantly harder to train.

**Future Work:**

We went to great lengths in order to try to use *Image-to-Markup Generation with Coarse-to-Fine Attention*’s evaluation metrics. However, we did not have enough time to properly implement their evaluation methods. Their code was built with Python, Perl, and Lua. We spent a lot of time trying to convert their code to all python. The metrics we hope to compute in the future are BLEU, Edit Distance, and Exact Match. That way we can compare our model with the previous effort in a more accurate manner.

Aside from evaluation metrics, we also want to conduct more rigorous investigation and thoroughly understand the behavior from the models, as we were quite lucky to have achieved

such performance. For example, we haven't even tried basic ideas like preprocessing the image (e.g. normalizing from a scale of [0, 255] to [0, 1] or [-1, 1], converting to grayscale, etc). Perhaps these changes would allow the worse performing models (especially the one with only an extra transformer encoder) to yield better results.

Finally, we also want to continue improving the website. One of the major improvements we ran out of time to work on was to optimize the model with TorchScript. The current state of the codebase still looks very "hacky" and there is much to improve when it comes to making sure it is easy to make revisions and add features. Setting up Docker containers and using CI/CD solutions may be time consuming, but increasing model portability and improving ease of use may pay off in the long run.