CS230

Automated LaTeX Code Generation from Handwritten Mathematical Expressions Category: Computer Vision

Jayaprakash Sundararaj
osjp@stanford.edu

Akhil Vyas
avyas21@stanford.edu

Benjamin Gonzalez-Maldonado
bengm@stanford.edu

Abstract

Converting mathematical expressions into LaTeX is challenging. In this paper, we explore using newer transformer based architectures for addressing the problem of converting handwritten/digital mathematical expression images into equivalent LaTeX code. We use the current state of the art CNN encoder and RNN decoder as a baseline for our experiments. We also investigate improvements to CNN-RNN architecture by replacing the CNN encoder with the ResNet50 model. Our experiments show that transformer architectures achieve a higher overall accuracy and BLEU scores along with lower Levenschtein scores compared to the baseline CNN/RNN architecture with room to achieve even better results with appropriate fine-tuning of model parameters.

1 Github Link

`https://github.com/osjayaprakash/deeplearning/tree/main`

2 Introduction

Converting handwritten mathematical expressions into digital formats is time consuming, specifically LaTeX code. Our goal is to train a ML model that is capable of encoding handwritten notes and converting to the source code seamlessly. The input to our algorithm is an image of a handwritten mathematical expression and the output is a sequence of tokens representing a LaTeX sequence. The challenge of our project requires the use of both computer vision and NLP techniques. We utilize an encoder-decoder architecture to encode the input image and decode it into a sequence.

3 Related work

? investigated a variety of methods like neural networks, CNNs, Random Forests, SVMs, OCR, CGrp, and SA. However, most state of the art methods utilize encoder-decoder architectures involving

CNNs and LSTM architectures like **?**. In recent works like **?**, both left-to-right and right-to-left decoders are utilized. The CNN-RNN architecture will serve as a baseline for our work.

Transformer architectures (**?**) are currently achieving the best results for NLP tasks . **?** introduced vision transformers which uses sequences of image patches to replace convolutions. We will leverage a vision transformer encoder and transformer decoder architecture and compare it to the baseline.

4 Dataset and Features

We will use the datasets from two main repositories: `Im2latex-100k` (**?**) and `Im2latex-230k` (**?**). The `Im2latex-100k` (**?**) dataset, available at Zenodo, contains 100,000 image-formula pairs. The `Im2latex-230k` (**?**) dataset, also known as `Im2latexv2`, contains 230,000 samples. It includes both OpenAI-generated and handwritten examples, further enhancing the diversity of the data. This dataset is available at Im2markup. The training data format is `<image file name> <formula id>`.

The dataset disk size is 849 MB. The images are gray scales with 50x200 pixels. The numbers of symbols (Figure 1) in the latex formulas vary from range varies from 1 to 150 symbols. Voabulary contains 540 symbols, refer Figure 7 and Figure 8 for the list of popular and least occurring symbols with their frequency.

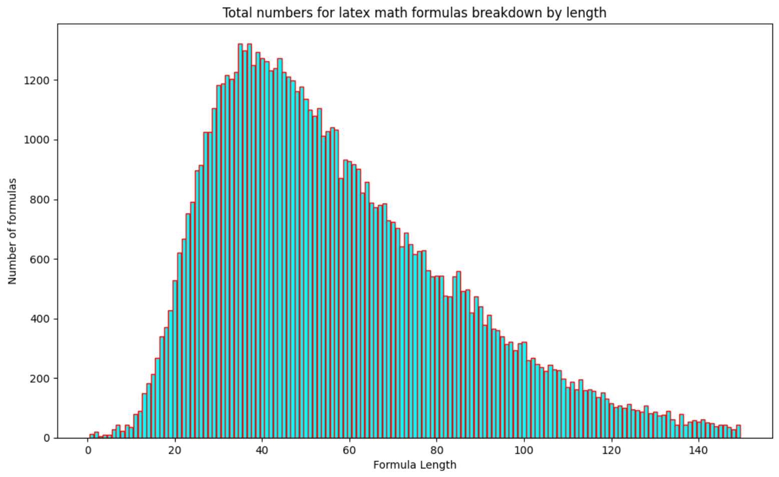


Figure 1: Formulas breakdown by length

5 Methods

5.1 CNN encoder and GRU/LSTM

As a baseline, We use the CNN Encoder to encode the image input of resized image (50x200) with 1 channel (greyscale). We use 3x3 convolutional filter followed by 2x2 max pooling layer. This previous block is repeated three times and followed fully connected layer.

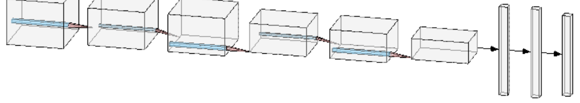


Figure 2: Encoder architecture consists of 3 convolution-max pooling blocks (50,200) -> (25,100) -> (12,5) which is flattened and fed into Dense layer (256 units)

During decoding, We compute the embedding for formula tokens and concatenated with image encoded embedding. The concatenation of image and token embedding fed into LSTM/GRU units, followed by fully connected network. The activation is softmax. Overall model architecture is:

$$\begin{tikzcd}
\boxed{\text{Input Image } I} \arrow[r] & \boxed{\text{CNN (Encoder)}} \arrow[r] & \boxed{\text{LSTM/GRU (Decoder)}} \arrow[r] & \boxed{\text{Output LaTeX Sequence } T}
\end{tikzcd}$$

5.2 LSTM with funetuning with pretrained Resnet50

Here we use the pretained ResNet50 model as a encoder (98Mb disk size). However, ResNet50 expects the image with fixed size 254x254 and 3 channels. Our input images are grey scale. So, we transform the input image to the ResNet50 input using `tf.keras.layers.Lambda(lambda x: tf.image.grayscale_to_rgb(x)`.

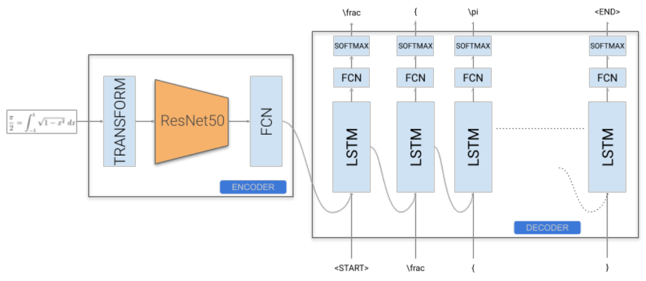


Figure 3: Pretrained ResNet50 Encoder with LSTM Decoder.

5.3 Vision transformer encoder and transformer decoder

5.3.1 Vision Transformer Encoder

The vision encoder leverages patches of the image. We create patches of 10 X 10. Since our images are of size 50 X 200, we have a total of 100 patches per image.

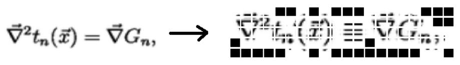


Figure 4: Original latex image and the generated patches

In the vision transformer encoder, these patches are taken and embedded linearly and added to the positional embeddings. That is fed into a standard transformer layer. We use 8 transformer layers for our architecture that have 4 attention heads and 2 layer multi-layer perceptron with 2048 and 1024 units.

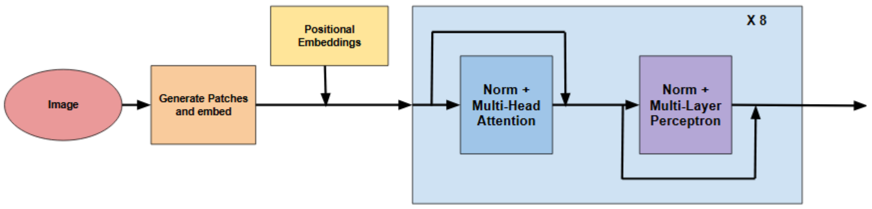


Image
Generate Patches and embed
Positional Embeddings
X 8
Norm + Multi-Head Attention
Norm + Multi-Layer Perceptron

Figure 5: Transformer encoder architecture

5.3.2 Vision Transformer Decoder

We use the standard transformer block for the decoder which uses cross-attention to find parts of the image to focus on and self-attention for the sequence generation. Our configuration uses 4 attention layers with 8 heads for both the self-attention and cross-attention components in each layer.

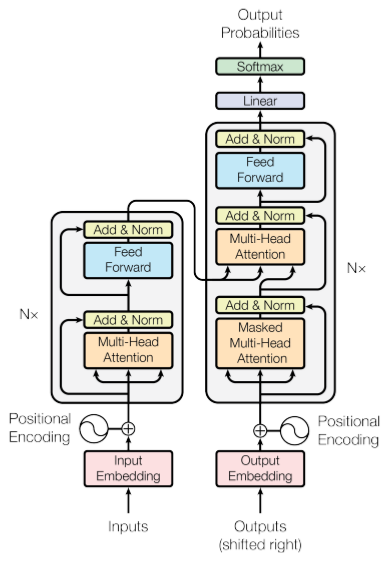


Figure 6: Transformer encoder architecture from **?**

6 Experiments

6.1 Setup/Hyperparameters/Metrics

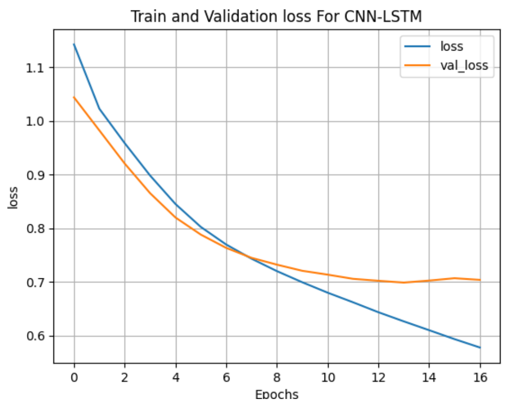
We use a single AWS G6.xlarge instance to train the models on 50,000 data points. The training time varies between 1 hr 30 mins and 2 hrs. We use early stopping with a patience of 10. We had a batch size of 128 for the CNN-RNN and ResNet-RNN, and a batch size of 64 for the transformer architecture which was primarily motivated by GPU memory constraints for the AWS instance. We used the default learning rate of 0.001 for the CNN-RNN and ResNet-RNN with the Adam optimizer, and a learning rate that decayed from 1e-4 to 1e-6 for the transformer architecture with the AdamW optimizer based on experimentation with different learning rates.

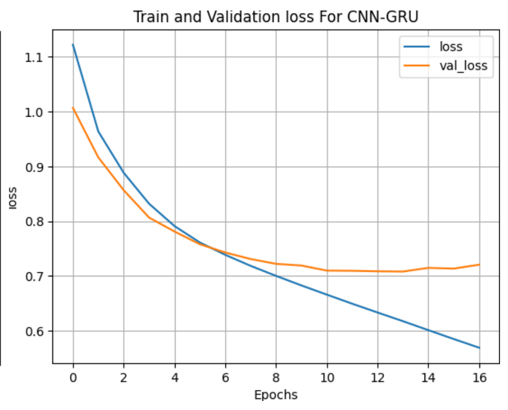
We compute the following metrics to compare the baseline and other methods:

- We measure the ‘sparse categorical loss‘ and accuracy which measures the loss/accuracy accross all tokens.
- We measure the masked loss and accuracy to measure the accuracy for the non-padded tokens (we pad our tokens to length 151 which is the max and this will only check the loss/accuracy for the tokens that are part of the label sequence)
- We measure the Levenshtein distance and BLEU-4 score for predicted sequences of a subset of the training set. These metrics were chosen in order to quantify closeness/correctness between sequences beyond a simple binary score that relies on exact matching.

6.2 CNN-RNN baseline

We explored using both LSTM/GRU for the decoder and the difference between the two were negligible. Here are the training curves with CNN - LSTM/GRU architectures:





Both models had 85% accuracy with GRU being slightly worse off. Due to this, we used the numbers from the LSTM decoder to compare against the other models.

6.3 Results

Architecture	Loss	Accuracy	Masked Loss	Masked Accuracy	BLEU	Levenshtein
CNN-RNN (LSTM)	0.6479	0.8470	1.6941	0.6008	0.4290	0.4016
ResNet-RNN	0.7811	0.8192	2.0793	0.5204	0.3723	0.4354
Transformer	0.5209	0.8738	1.4722	0.6417	0.5575	0.3546

We can see that the transformer architecture gets significantly lower loss and Levenshtein score, and higher accuracy and BLEU score compared to the baseline CNN-RNN model and ResNet-RNN model.

7 Conclusion/Future Work

We can see that overall the transformer architecture outperformed the vanilla CNN-RNN architecture on all measured metrics. We surmise that this is due to the combination of the algorithm utilizing positional embeddings along with attention mechanisms for both the encoder and decoder. Given more time, we would have mainly focused on experimenting with various transformer architecture configurations (by changing number of layers, attention heads, patch size for the vision transformer encoder etc.). We also would have looked at training more complex models with more data. For this paper, we used 50,000 data points with appropriate model sizes due to GPU memory constraints on the AWS instances we could use.

8 Contributions

Jayaprakash Sundararaj: Initial report, researching the dataset and existing methods. Implementing the full CNN and LSTM as a baseline. Extending to pre-trained ResNet50 model with finetuning.

Akhil: Ideation, AWS/GPU setup, Extending to CNN + GRU as a baseline, vision transformer encoder + transformer decoder model, masked loss and accuracy.

Ben: Looked into potential final accuracy metrics, Implementing the Levenshtein and BLEU-4 metrics specific to models based on prediction outputs.

9 Appendix

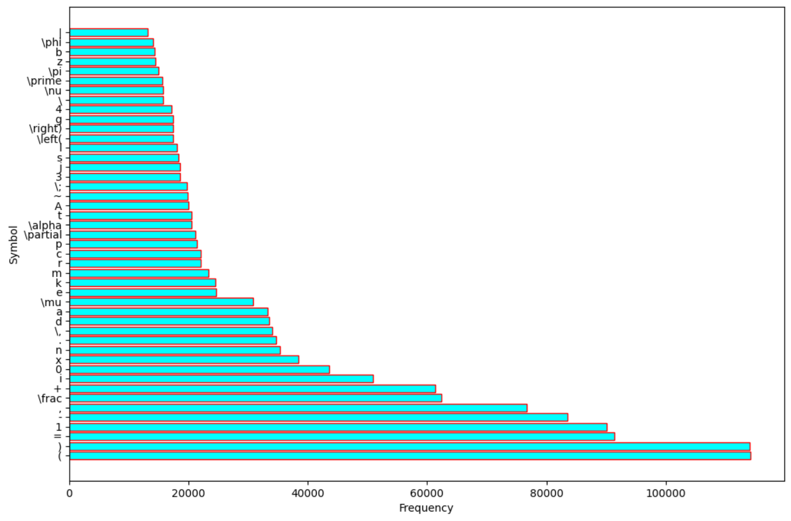


Figure 7: Dataset: Most popular symbols and frequencies.

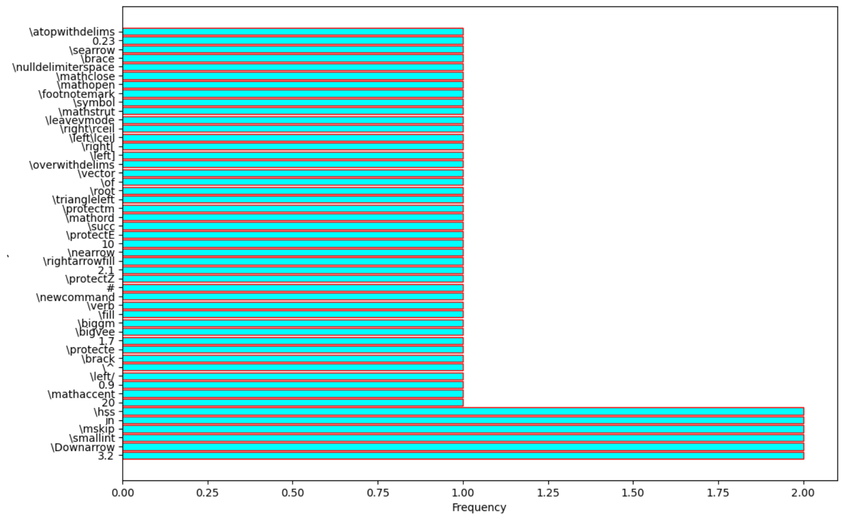


Figure 8: Dataset: Least popular symbols and frequencies.