
Neural Networks for Image to LaTeX

Yuzhe Bai

Department of Mathematics
University of California, San Diego
La Jolla, CA 92093
yub017@ucsd.edu

Bochao Kong

Department of Mathematics
University of California, San Diego
La Jolla, CA 92093
bokong@ucsd.edu

Zichen He

Department of Mathematics
University of California, San Diego
La Jolla, CA 92093
z7he@ucsd.edu

Cameron Cinel

Department of Mathematics
University of California, San Diego
La Jolla, CA 92093
ccinel@ucsd.edu

Rishabh Bhattacharya

Department of Mechanical and Aerospace Engineering
University of California, San Diego
La Jolla, CA 92093
ribhattacharya@ucsd.edu

Madeleine C. Kerr

Scripps Institution of Oceanography
University of California, San Diego
La Jolla, CA 92093
mkerr@ucsd.edu

Shrey Kansal

Department of Mechanical and Aerospace Engineering
University of California, San Diego
La Jolla, CA 92093
skansal@ucsd.edu

Abstract

In this study, we investigated four popular neural network architectures for solving the image-to-LaTeX problem using the Image2Latex 140K dataset [6]. Our first architecture, CALSTM, was based on [6], and achieved a BLEU-1 score of 77.76%. Our second architecture, a ResNet-Transformer, produced a higher BLEU-1 score when we changed the position encoding method to 90.4%. Then, we used a visual transformer (ViT) to achieve a BLEU-1 score of 91.5%. The last model we built is ResNet with Global Context, which reaches BLEU-1 score 83.5%.

Based on these results, we selected the ResNet-Transformer architecture as the best candidate and improved its generalization through data augmentation. We are able to get 93.3% BLEU-1 with data augmentation. Our model can now handle real formula images obtained from screenshots.

We produced visualizations of the gradient and attention maps to understand the explainability of the models. Our findings suggest that the models behave similarly to us.

1 Introduction

Image to Latex code is a special task in optical character recognition (OCR). The ability to automatically convert handwritten or printed mathematical symbols and equations into \LaTeX code can greatly improve accessibility, save time and effort, and enhance the accuracy of reviewing and reproducing

scientific documents. Neural networks have shown promise in various computer vision tasks, and the image-to-LaTeX problem is no exception. However, the problem presents unique challenges, including ambiguity in the mapping between the image and the \LaTeX code and the sensitivity of \LaTeX to the relative positions of symbols.

In this project, we investigate four popular neural network architectures for solving the image-to-LaTeX problem using the Image2Latex 140K dataset [6]. Our first architecture, CALSTM, is based on the model proposed in [6], and we achieved a BLEU-1 score of 77.76% using baseline model, and we implemented beam search which gate a BLEU-1 of 77.46%.

We then explored the use of a ResNet-Transformer architecture proposed/implemented in [7] and repo here. The baseline model can give 85.3% BLEU-1 score, which is better than the CALSTM. We further change the position encoding method and it improves BLEU-1 to 90.4% with 28 million parameters.

In the third architecture, we used a visual transformer (ViT) architecture to achieve a BLEU-1 score of 91.5% with about 40 million parameters indicating its potential for further improvement.

Lastly, we tried to modify the ResNet-Transformer and obtained ResNet-Global Context-Transformer model, which can have BLEU-1 score as 83.5% with about 17 million parameters.

The best performance ResNet-Transformer model not has perfect BLEU scores, but its performance is actually nearly perfect. The only visible "mistakes" it would make are two types: synonymous and extra braces. For example:

`\rightarrow = \to \Rightarrow`, `{any formula} = any formula`.

For the synonymous problem, although `\rightarrow` and `\to` are rendered the same, but there are still conventional usage differences, `\to` is preferred when we use arrow define a function, and `\rightarrow` is preferred when build commutative diagrams. We hope the model can learn these difference and write code more close to humans. So we're not mapping all the synonymous to the standard choices. For the extra braces problem, the model prefer to do simplification and we don't think that will be an issue.

Now the models have good performance on the dataset itself. But when it comes to a real formula image, things are more complicated. We can't guarantee the image size, resolution/dpi are all aligned with the ones in the dataset. We took the best performance model and used data augmentation technique to improve the generalization. Surprisingly we can get 93.3% BLEU-1 score from 6-layer decoder model with random scaling added. As a result, we produce a model that can handle real world screen-shot images, although usually with some minor mistakes.

Our models all has built-in attention mechanism, and we investigated the attention maps for CALSTM and ResNet-Transformers, the visualization images are shown in last section. The models are (mostly) able to pay attention to the correct places just as we human do.

2 Related Works

The problem of recognizing math formulas from images has a long history dating back to 1967 [1] when traditional OCR methods were first applied. However, these methods did not perform well in recognizing math symbols as they can appear in various forms such as sub/super scripts, matrices, and nested structures. One of the most successful traditional systems is INFTY [9], which converts an image to markup languages such as \LaTeX .

The use of neural networks to solve the image-to- \LaTeX problem began with the work of Deng et al. [3]. Their dataset is still widely used in the field. A milestone work was presented in [6], where the authors created a larger and better dataset named Image2Latex 140K and achieved state-of-the-art performance with an 89% BLEU score. Since then, the use of transformer-based architectures has become popular. The ResNet-Transformer architecture was proposed in [7] and achieved a BLEU score of 93%. We modified the 2D position encoding they proposed and, even with random scaling as data augmentation, we can achieved a BLEU score of 93.3%.

We also use a ViT inspired transformer encoder-decoder model for this formula-generation task. Note that the original paper [4] was implemented for image classification, thus we had to make some changes to the architecture, which have been detailed below.

The last model, ResNet with Global Context - Transformer, was based on the paper [5], where the authors had implemented a deep ResNet34 with Global Context Block based encoder, which also included a transformer in the end. We removed the transformer layer and implemented a much shallower ResNet18 suiting our use case. While the authors achieved a BLEU-1 score of about 89%, our score of 83.5% seems respectable.

Most of the architectures developed for this task can be generalized to recognize handwritten formulas. A survey comparing different approaches in this direction can be found in [2]. However, due to time limitations, we did not test our models' generalization ability to handwritten formulas. Instead, we focused on data augmentation techniques to improve the models' performance on reasonably clean screenshots, similar to the commercial software "Mathpix".

3 Dataset

To train and test the models, we used the Image2Latex140k dataset from [6]. In this dataset, most preprocesses are already finished. The tokens are extracted and put in a vocabulary file with indices associated. Moreover, the formulas in the dataset have been categorized into several categories based on their length. The formulas in the same category are padded into the same length, so that for each formula, only a small amount of padding is needed. The distribution of the padded lengths in the training set are as follows:

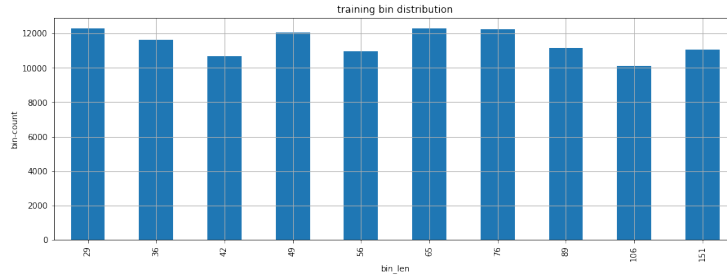


Figure 1: Training Bin Sizes

This completed preprocess greatly reduced the amount of time required for training as we don't need to pad almost every formula to the length of the longest one in each batch. In order to utilize this convenient structure we choose to consider each one of the categories a sub-dataset and choose them in a random order each time and then apply shuffle inside each category.

However, we still need to process the images that contains the formulas a little bit. For each of the images, it contains three color channels and one alpha channel. Since the picture is has only white and black colors, these four channels literally contain the exact same data. Hence, we extract the data of one of the channels and change that into a tensor that only has value 0 (which means black) and 255 (which means white). All the formula images are properly cropped, and the dataset contains multi-line large formulas as well, below are three examples:

$$\bar{\partial}_i \tilde{s} = 0 \quad .$$

Figure 2: A short formula

$$\langle \varphi^2(x) \rangle_b \approx -\frac{\Gamma^{-2}(\nu_0 + 1)}{2^{2\nu_0} \pi a^{D-1} S_D \sigma^{D-1}} \left(\frac{r}{a}\right)^{2\nu_0-n} \int_{ma}^{\infty} dz \frac{z^{2\nu_0+1}}{\sqrt{z^2 - m^2 a^2}} \frac{\bar{K}_{\nu_0}(z)}{I_{\nu_0}(z)}, \quad r \rightarrow 0,$$

Figure 3: A long one line formula

$$\begin{aligned} h(a,b) &= \frac{1}{a-b} \ln \frac{a}{b}, \\ B(a,b) &= \frac{1}{(a-b)^2} \left[\frac{1}{2}(a+b) - \frac{ab}{a-b} \ln \frac{a}{b} \right], \\ p_b(a,b) &= f(a,b) - 2e_b \sin^2 \theta_W (a-b) g(a,b), \\ p_t(a,b) &= f(a,b) + 2e_t \sin^2 \theta_W (a-b) g(a,b), \\ g(a,b) &= \frac{1}{(a-b)^2} \left[2 - \frac{a+b}{a-b} \ln \frac{a}{b} \right], \\ f(a,b) &= \frac{-1}{a-b} \left[1 - \frac{b}{a-b} \ln \frac{a}{b} \right]. \end{aligned}$$

Figure 4: A multi-line formula

As all the images are cropped to different sizes, we need to pad the images, such that the formulas are in the middle. We choose to pad into the same size that is 128×1088 which is recommended by the dataset author. But by doing that, we discard the "tall" formulas like the one in Figure4, so this limits our model's ability to understand multi-line formulas.

Initially, that's all preprocesses done to the dataset. The best ResNet-Transformer model trained on this dataset gives us a BLEU-1 score more than 90%. However, when we try to use this model to translate an image we take to L^AT_EX code, it feels miserably. It turns out that the model only have good behavior on images whose formulas are in a specific font size and are in the middle of the images. In order to make the models work better generally, we decided to apply several transforms, including dilation, adding noise and add blurring, to each of the images.

4 Methods

4.1 CALSTM

We wrote a PyTorch implementation of the Conditional Attention Long Short Term Memory (CALSTM) model from [6]. This model consists of two main parts: a fully convolution encoder and a recurrent decoder with attention. The precise details of the structure can be seen in 5. We trained our model for 30 epochs with a learning rate of 5×10^{-5} using the Adam optimizer with a weight decay of 5×10^{-5} , just as in [6].

The encoder consists of 12 layers, alternating convolutional layers of kernel size 3 and stride 1 with maxpool layers of kernel size and stride 2. The activation for each convolutional layer is tanh. Additionally, the output is flattened from dimensions $4 \times 34 \times 512$ to 136×512 .

The decoder layer consists of 5 sub-parts: the Init State Model, the Attention Model, a word embedding, a stacked LSTM, and a fully connected output layer.

4.1.1 Init State Model

The Init State model consists of 2 fully connected layers with 100 and 6000 hidden units, respectively, that takes in the encoded image features and outputs the initial hidden states of the LSTM. The activation for each of these fully connected layers is tanh.

4.1.2 Attention Model

The Attention model consists of 3 fully connected layers with 256, 128, and 136 hidden units, respectively, and using tanh activation. For each of our 136 receptive fields, the 512 feature channels are passed through the attention model and then softmaxed to give weights to each of the receptive fields at time step t , which we denote by α_t . The encoded image features a is then multiplied by the

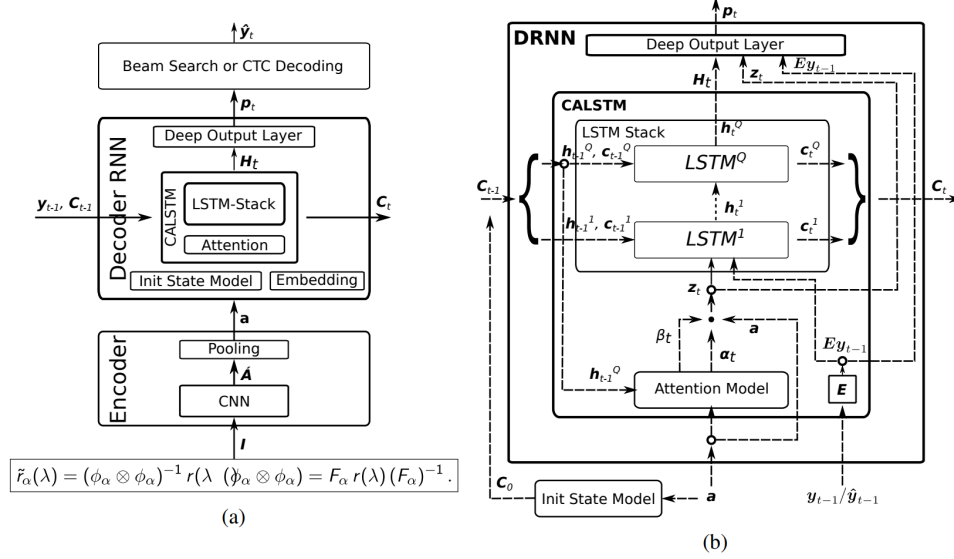


Figure 5: Pictorial representation of the CALSTM model from [6]. (a) Shows the entire model structure, with both encoder and decoder at a single time step. (b) Shows the decoder structure in more detail, with \vec{a} being the encoded image features, flattened from $H \times W \times D$ to $L \times D$.

weights α_t to give the conditioned image features z_t . Figure 5 shows an additional scaling parameter β_t , however the CALSTM model used in [6] does not make use of this factor. Additionally, the weights between each of 136 attention models share their weights. The inputs to the Attention model are a concatenation of the encoded image features with the hidden state of the final LSTM layer for the previous timestep. For the initial time step, this hidden state is the one output by the Init State Model.

4.1.3 Embedding

We used a one-hot encoding followed by a linear embedding to embed our tokenized words into \mathbb{R}^{300} .

4.1.4 LSTM

The LSTM component consists of 2 stacked LSTM layers, each of 1500 units. At each time step, the LSTM takes in a concatenation of the conditioned image features z_t and the embedded word Ey_{t-1} of the previous timestep. During training, we used teacher forcing so y_{t-1} is previous word of the ground truth. During inferencing, we did not use teacher forcing, and so y_{t-1} is the previous output word from the model. We chose 2 LSTM of 1500 hidden units at that was the optimal configuration from [6].

4.1.5 Deep Output Layer

The deep output layer consists of 3 fully connected layers with the first two having 358 hidden units and tanh activation, and the last layer having the number of units equal to the vocab size with softmax activation. The deep output layer takes in a concatenation of the output of the LSTM H_t , the conditioned features z_t and the previous embedded word Ey_{t-1} . The output is then a probability of the possible word choices, which were then chosen either via a greedy algorithm or a beam search.

4.2 ResNet-Transformer

4.2.1 Architecture

This model uses standard encoder-decoder structure. The encoder is a ResNet18 up to layer3, and we use this network to extract 256 features from the image. Moreover, since we only extract one channel from each image, instead of a convolution layer with 3 input feature, we used a convolution layer

with one input feature as our first layer. Then we apply 2D-position encoding introduced in [7] to maintain the spatial information of the features. Then we flatten the features and pass it to a standard transformer decoder. The architecture can be summarized in the following diagram in [7]:

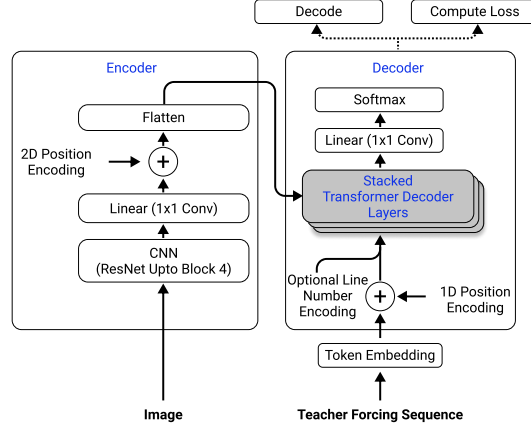


Figure 6: ResNet-Transformer Model

Most of the parts in the network are standard architecture. We use the ResNet-18 without pretrained weights up-to layer 3, our input images are one channelled, so we modified the first layer of the ResNet-18 to be a 1×1 convolution that take in one channel and output 64 channels.

4.2.2 2D Position Encoding

The only worth noticing thing is the 2D position encoding layer, which is a variation of the usual position encoding in [10]. The formulas proposed in [7] as:

$$\begin{aligned}
 PE(y, 2i) &= \sin \left(y / 10000^{2i/d_{\text{model}}} \right) \\
 PE(y, 2i + 1) &= \cos \left(y / 10000^{2i/d_{\text{model}}} \right) \\
 PE(x, d_{\text{model}}/2 + 2i) &= \sin \left(x / 10000^{2i/d_{\text{model}}} \right) \\
 PE(x, d_{\text{model}}/2 + 2i + 1) &= \cos \left(x / 10000^{2i/d_{\text{model}}} \right) \\
 i &\in [0, d_{\text{model}}/4)
 \end{aligned}$$

The main idea of this 2D encoding is do a standard attention is all you need position encoding twice, where we use half of the features to encode the horizontal spatial information and half of the features to encode the vertical spatial information. And the implementation here follows the encoding method but has a minor bug.

We implemented the above 2D position encoding as well, we're able to get 85.27% BLEU-1 score from it. But we believe every feature should be treated equally, so in our implementation, we did vertical and horizontal encoding for all features and concatenate. With similar number of features and input dimension of the transformer, we can improve the BLEU-1 score to 90.4%.

4.2.3 Loss Function

For the loss function, we used the CrossEntropyLoss. But we need to compare predicted latex codes with target latex codes (teacher). In teacher forcing, the input is the teacher, and i -th element from input will output i -th element in prediction which should match with $(i + 1)$ -th element in target. For example, the first string in target is ' $< bos >$ ' and we expect our model will output the second string in target. Hence, we have to shift target by one in loss function. However, we did not do it in the code of the first version. The model was trained to directly output the input and made no prediction.

4.3 ViT

We wrote a PyTorch implementation of Visual Transformers (ViT) [4]. Even though the original work was designed for image classification, we have modified some parts of it, while keeping many parts same. It majorly consists of a patch embedding layer, followed by an encoder and a decoder.

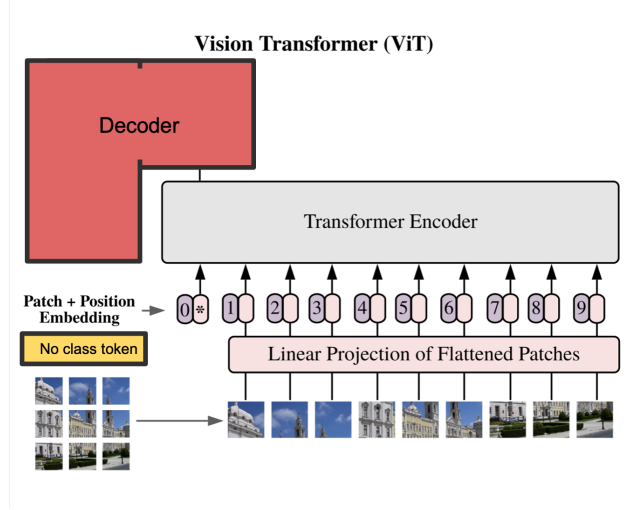


Figure 7: ViT architecture

The encoder architecture in ViT is designed to capture the spatial relationships between different parts of the input image by using self-attention mechanisms, while also leveraging the powerful representation learning capabilities of the transformer architecture.

The decoder takes the encoder representation and generates an output sequence that is appropriate for the given task. The decoder can be viewed as a language model that generates a sequence of tokens, such as words, based on the encoder representation. In our case, we intend to generate the \LaTeX formula for the input math equation.

4.3.1 Patch Embedding

We substituted the path embedding convolution layer with a linear fully connected layer. In the ViT architecture, the image is first divided into a grid of fixed non-overlapping patches and then each layer is embedded into a higher dimensional space. The patch embedding method transforms the raw pixel values of the input image into a sequence of feature vectors, which can be processed by the transformer layers. The reason for using linear layer instead of a convolution layer is that the latter is better suited for capturing spatial information, rather than sequential information, which is predominant in this task of converting mathematical formula images to latex code.

Post linearly projecting the input image into a feature vector, we prepend a start token to it as a parameter. The start token is an essential parameter for the vision transformer, as it indicates the start position of the generated sequence of patches. Now, a positional embedding is added to this resultant feature vector for indicating the position of each patch to the transformer network.

4.3.2 Encoder

The Encoder block is derived from the implementation in [10], with its PyTorch implementation here.

The encoder architecture in ViT is composed of multiple layers of self-attention and feedforward networks, similar to the architecture used in language transformers like BERT or GPT. However, unlike language transformers that process sequences of tokens, the input to the ViT encoder is a sequence of image patches.

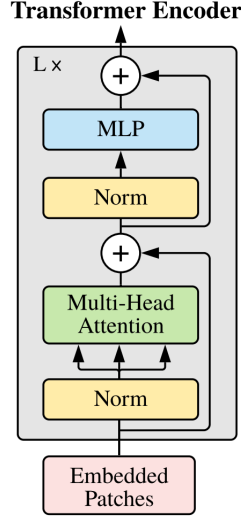


Figure 8: Transformer encoder; L is the depth of the encoder

Specifically, the ViT encoder starts by dividing the input image into a fixed number of non-overlapping patches (16×16 in our case), and flattening each patch into a 1D vector. These patches are then projected into a sequence of embeddings, which serve as the input to the transformer encoder.

In each transformer layer, the embeddings are first processed by a multi-head self-attention mechanism, which allows the model to attend to different parts of the input image at different scales. The outputs of the self-attention mechanism are then passed through a feedforward network, which applies a non-linear transformation to each embedding. This process is repeated for multiple layers, allowing the model to learn increasingly complex representations of the input image.

4.3.3 Decoder

The Decoder block is derived from the implementation in [10], with its PyTorch implementation here. We have modified the decoder to incorporate a word positional encoding. Since transformers are inherently parallelized (process inputs in parallel), positional encoding is used to provide the model with information about the relative position of each token in the sequence, allowing it to generate output tokens in the correct order.

The decoder layer in the transformer architecture includes three sub-layers: masked multi-head self-attention, multi-head attention over encoder outputs, and a feedforward network. These sub-layers allow the decoder to attend to the input sequence and generate an output sequence. Residual connections and layer normalization are included in each sub-layer to enhance training stability.

4.4 ResNet with Global Context - Transformer

Global Context (GC) Networks were originally developed for image classification to model the global spatial dependencies between different objects in the input image. Since, our task at hand is sensitive to spatial dependencies between different characters of the formula, it seemed like a good idea to implement. The GC block, Fig 9, was integrated with a ResNet18 backbone, while the decoder remained the same (masked transformer based decoder).

The GC block is basically implemented in 3 steps:

1. **Global Attention Pooling:**

This step captures the attention weights for global context modeling. It uses a 1×1 convolution layer to render the input feature maps to just 1 channel.

2. **Bottleneck Transform:**

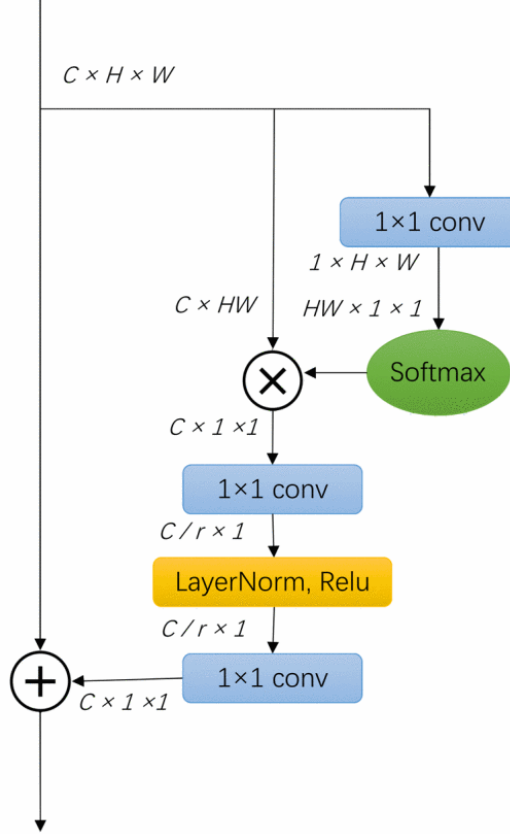


Figure 9: Global Context Block

This step captures the interdependence between different channels using the global attention pooling layer. This is done by matrix multiplying the unrolled input feature map with the unrolled single channel output of global attention pooling layer.

3. Feature Aggregation:

Finally, the global context features are aggregated for each position in the feature map by element-wise adding the output of previous layer with the input feature map.

4.4.1 Encoder

The Global Context blocks are integrated with untrained ResNet18 layers 2 and 3 for feature extraction. A bottleneck layer is implemented as well before we embed the feature maps with $\sin - \cos$ positional encoding. After each block consisting of GC and ResNet layer, the feature map is downsized using a `maxpool` layer, which also helps to capture the spatial dependencies in the feature map. The bottleneck layer constraints the number of channels of feature map to be half of the desired decoder embedding size. These channels are the concatenated, which enables us to assign a \sin and \cos value to each position.

We did not implement a hybrid ViT network, which would take the ResNet features as input due to computational constraints. Although, we do expect it to perform better, there is no metric to pass a judgement as to how it might perform. Furthermore, ResNet34 would also improve the encoder's performance, but we would need to carefully tend to the issue of overfitting and vanishing gradients (prominent in deep networks).

We used the same position encoding and loss function as implemented with the ResNet-Transformer model in section 4.2. Furthermore, we used a multi-step learning rate scheduler and randomly scaled the images to account for variance in input image sizes. The transformer based masked decoder used to predict the $\text{L}^2\text{E}^2\text{X}$ formula remained the same as in the case of ResNet-Transformer and ViT models.

4.5 Beam Search Implementation

Beam search is a useful algorithm in sequential deep learning applications. The algorithm should match or outperform the greedy search algorithm when selecting outputs after the soft-max layer in the final steps of testing auto-encoder architectures such as the ones we use. This tool takes the initial sequential output from the decoder layer and keeps track of the previous words used to produce a sentence output that achieves the best possible performance on an optimal sub-tree of the entire tree of possible captions for a given trained decoder and decoder input.

The overarching idea behind beam search is to follow the B best paths through the sequential output, starting with the "bos" (beginning of sentence) symbol and after the maximum LaTeX caption length is achieved, or after all B best paths reach the "eos" symbol (i.e. end-of-sentence), to output the best caption formed by the remaining B paths. B is a hyper-parameter called the "beam-width" and it is the number of paths that the algorithm tracks over the course of the sequential output.

Algorithm 1 Beam Search over batches in auto-encoder architecture

a is the batch of LaTeX images
 A is batch size
 B is beam width
 L is the max length for a latex caption
 C is the total number of latex symbols
 \mathbf{X} is the output of Encoder(a); size: $[A \times \text{Encoder output size}]$.

Initialize **prevProbs** tensor, size $[A \times B]$
Initialize **outWords** tensor, size $[B \times A \times (L + 1)]$ with all $B \times A$ vectors starting with "bos".
Initialize **hasEnded** tensor, size: $[A \times B]$ to track continuation of captions
for pos in range(L) **do**
 $N \leftarrow \text{pos} + 1$
 Initialize **allPathProbs**, **allPathWords**, and **allPathAdds** sizes: $[A \times B \times B]$
 for b in range(B) **do**
 ended? \leftarrow **hasEnded** $[:,b]$, beam b value across all A batches.
 $y \leftarrow$ **outWords** $[b]$ for all A batches and only the first N values for each.
 logits \leftarrow output of Decoder(\mathbf{X}, y), size: $[A \times N \times C]$
 sort **logits** in descending order along dimension 2 (with length C), save **wordIdxs**.
 get (**Top B logits** and **Top B wordIdxs**).
 assign these to element b of **allPathProbs** and **allPathWords** for each batch
 assign **ended?** value to element b of **allPathAdds** for each batch.
 end for
 flatten **allPathAdds**, **allPathWords**, **allPathProbs** along dim 1, sizes: $[A \times B^2]$.
 sort flattened allPaths tensors with **allPathProbsFlat** along dim 1.
 select top B for all 3 tensors and get size $[A \times B]$
 these are **topWords**, **topProbs**, and **topEnded**.
 for a_i in range(A) **do**
 for b_i in range(B) **do**
 word \leftarrow **topWords** $[a_i][b_i]$
 ended \leftarrow **topEnded** $[a_i][b_i]$
 if **ended** is True **then** continue
 else **outWords** $[b_i][a_i][N] \leftarrow$ **word**
 end if
 if **word** is "eos" **then** **hasEnded** $[a_i][b_i] \leftarrow$ True
 end if
 end for
 end for
 update **PrevProbs** with **topProbs**, the new previous probs.
 if all(**hasEnded**)=True **then** break
 end if
end for
bestCaption \leftarrow the best $(L + 1)$ caption from **outWords** using the argmax of **prevProbs**.
return **bestCaption**

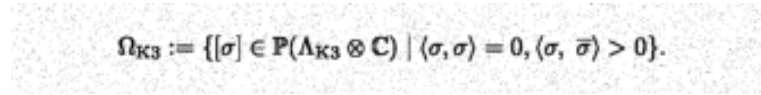
4.6 Generalization

The three architectures we implemented can all have good performance on the dataset itself. And we decide to take the best performance model and try apply it to the real world situation. Note that the model is trained without data augmentation at all, so all its learning are based on the images from the dataset. We took a screen shot of a simple formula: The model fails to give the answer if we just use

$$(1 + X_1)^2 + (2 - X_2)^2 = 4.$$

the screen shot as the input. The main issue is the size of this screen shot appears significantly **bigger** than the formula images in the dataset.

So to really make the model usable in real world, we use data augmentation and retrain the model. We use randomly scale the image from dataset from 0.8 to 1.4 as our first data augmentation, the ResNet-Transformer can converge, see results section for more details. Then we further added on the salt-pepper noise to to the randomly scaled output:



$$\Omega_{K3} := \{[\sigma] \in \mathcal{P}(\Lambda_{K3} \otimes \mathbb{C}) \mid \langle \sigma, \sigma \rangle = 0, \langle \sigma, \bar{\sigma} \rangle > 0\}.$$

Figure 10: Salt-Pepper Random Noised

However this reduced the model's performance and cannot be improved by change model size and other hyperparameter tuning method. The reason for that should be the problem is pixel-sensitive, for example:

$$(a - b)^c \neq (a - b)^e.$$

On the superscript, the letter c and e may only differ by few pixels, the salt-pepper noise would very likely confuse the model in similar situations.

And we also add preprocess for the images outside of the dataset, we crop the formula image according to the four corners and resize according to the font thickness.

5 Results

5.1 CALSTM

After training, our CALSTM model achieved a loss of 6.154×10^{-3} , a BLEU-1 score of 77.76%, and a BLEU-4 score of 56.54% on our test dataset. A chart of loss of the training epochs can be seen in Figure 11. Unfortunately, due to time constraints on training the full model (approximately 20 minutes per epoch), we were only able to train the model for approximately 1/3 as long as in [6] and thus did not achieve similar level results. Additionally, we were unable to train multiple versions of the same model with different hyperparameters for similar reasons.

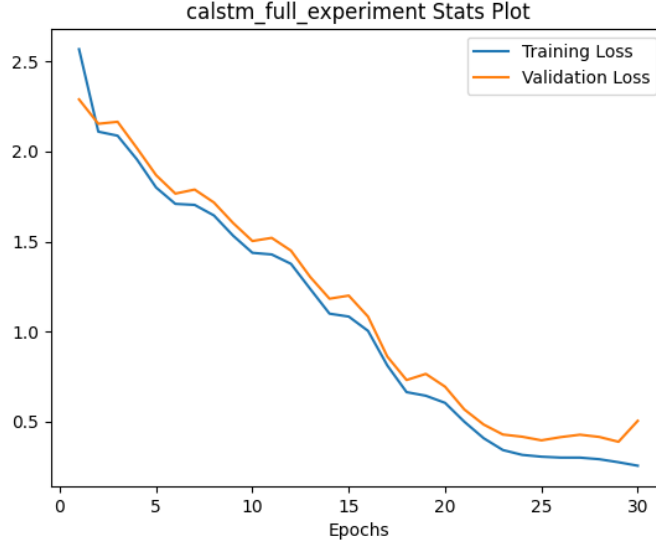


Figure 11: Loss plots for the CALSTM model.

We have visualized the attention given by the model in Figures 12, 13, and 14. Figure 12 represents the model effectively using attention, paying attention mostly to a single receptively for a single symbol such as 1 or a collection of nearby receptive fields for symbols that indicate a relation between other symbols such as `\frac`.

Figure 13 is an example of the attention mechanism failing. In this case, the model needs to generate a 3×3 matrix using the `\begin{array}` command. In `\LaTeX`, the argument for the `\begin{array}` environment is a collection of letters and separators indicating the number of columns, their alignment, and the separators between columns. In the case of a matrix, the most common letter used is `c` (for centering) used as many times as there are columns with only a space separator. In the attention for each the `c`'s, the attention only seems to be able to weight two of the columns of the time, causing the matrix to instead only generated a 3×2 matrix. While the model can effectively pay attention to the symbols within the matrix, it seems to cut off the third column after already creating the environment for a 2 column matrix. Occasionally, in other examples, the model will continue to write the inside of the matrix as normal, leading to a compile error in the generated `\LaTeX`code due to mismatching column numbers. Most of the compile errors in generated code are of this form after sufficient training.

Figure 14 indicates another failure of the attention mechanism, this time due to nearly identical repeated chains of symbols. In this example, we have the chain of characters $D_{i \rightarrow (J/\psi, \chi_i)}$ and $D_{j \rightarrow (J/\psi, \chi_i)}$ in the equation at different locations. The attention heat map shows approximately equal weighting being used as the model generates the text for each of the chain of symbols. However, instead of outputting z as it should for the first chain of symbols, it instead outputs y for the second chain, before moving back to the `=` and then generating the rest of the equation. This is likely due to the lack of positional encoding in the CALSTM model, with the model not being able to differentiate between receptive fields if locally, i.e. the field itself and the adjacent fields, are nearly, if not exactly,

identical. In this example, the model was able to continue generating the rest of the equation, though in other examples the model would often skip the equation between the two repeated chains entirely.

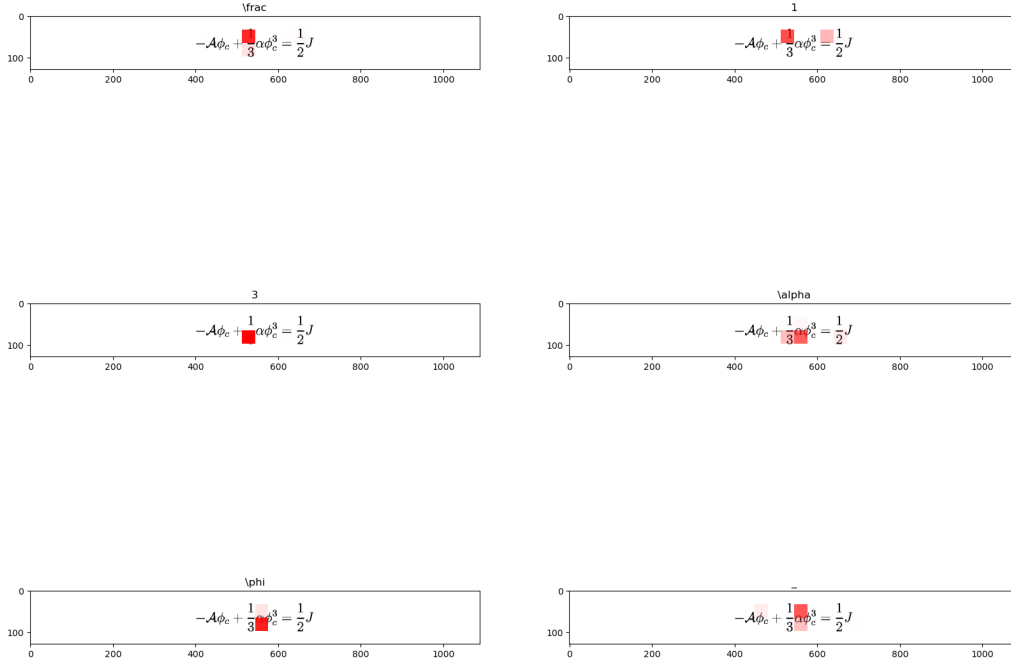


Figure 12: Visualization of the condition image features weights α_t . The red squares indicate receptive fields that the model is paying attention to, with darker red indicating higher weights. This is an example of the model effectively paying attention to various symbols.

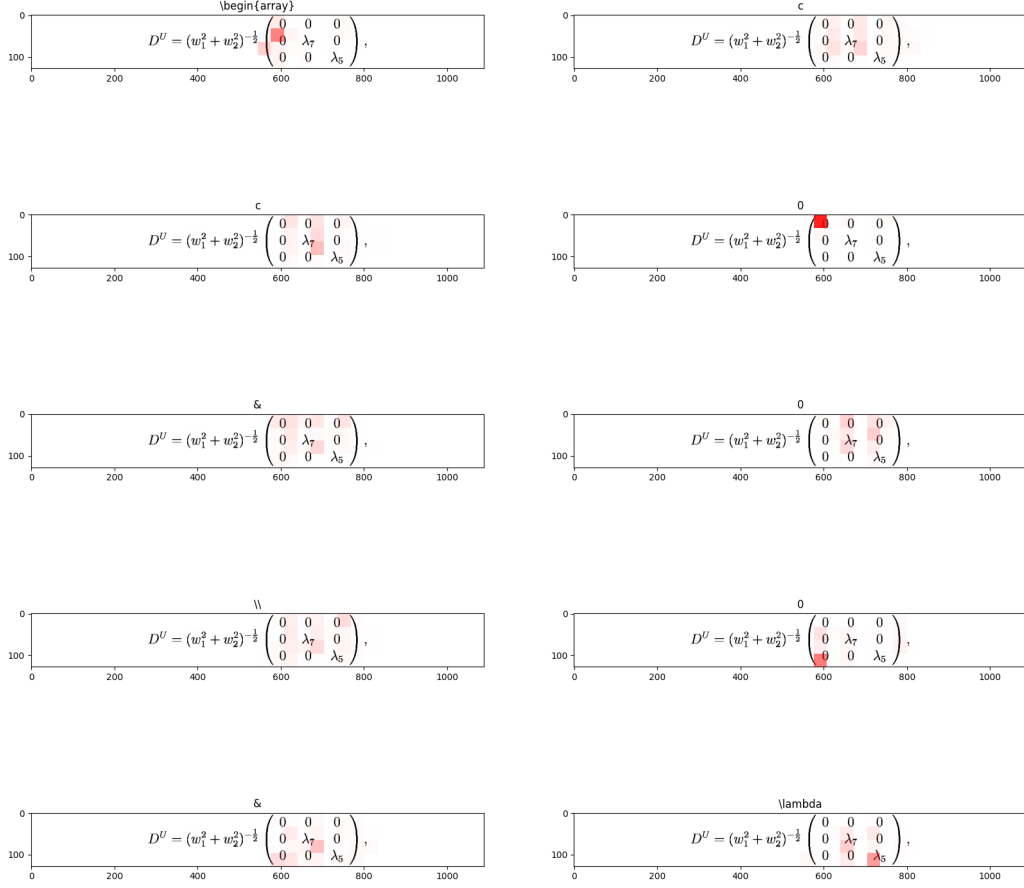


Figure 13: Visualization of the condition image features weights α_t . The red squares indicate receptive fields that the model is paying attention to, with darker red indicating higher weights. This is an example of attention failing due to needing to paying attention to numerous receptive fields.

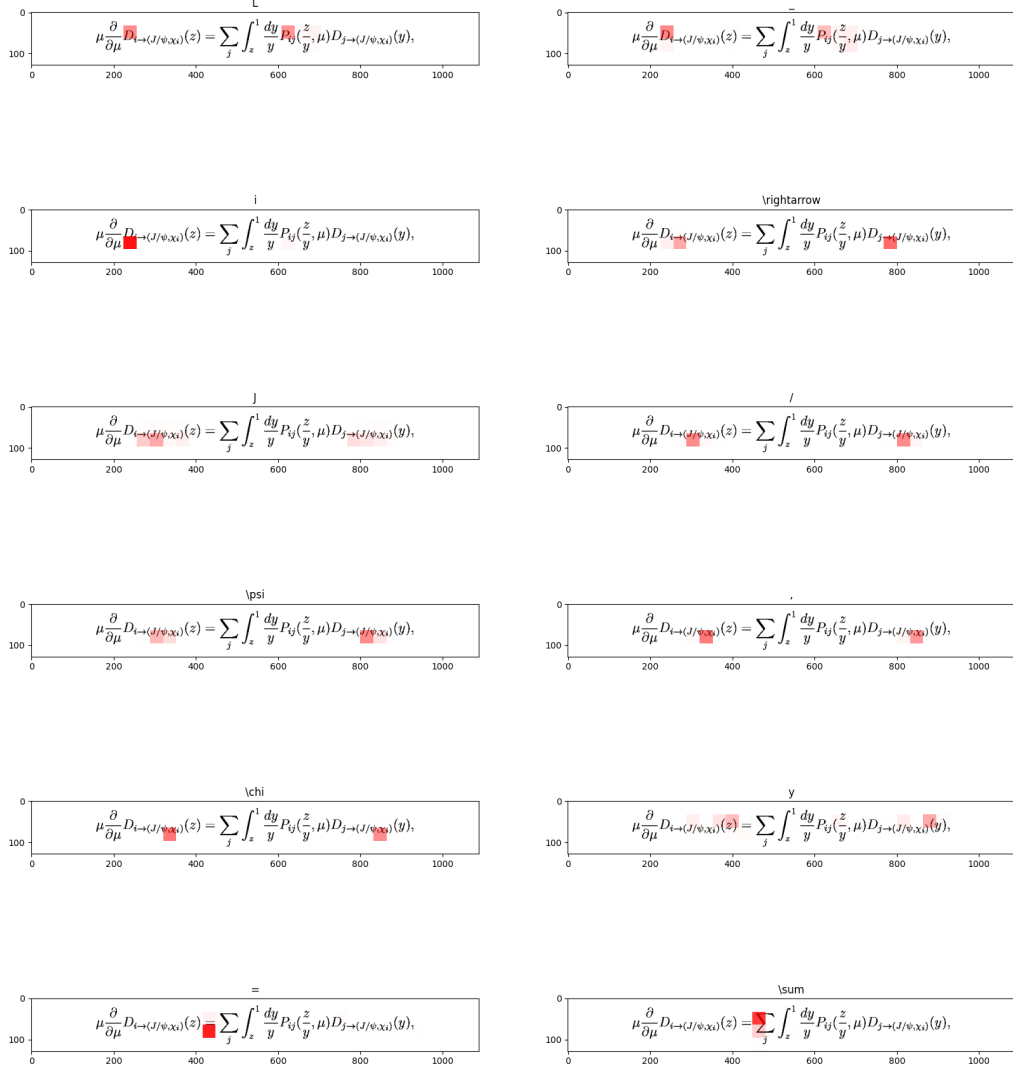


Figure 14: Visualization of the condition image features weights α_t . The red squares indicate receptive fields that the model is paying attention to, with darker red indicating higher weights. This is an example of the model simultaneously paying attention to repeated symbols.

5.2 ResNet-Transformer

5.2.1 Baseline Model

We achieved the BLUE-1 85.27% and BLUE-4 70.4%. The loss plot is shown below:

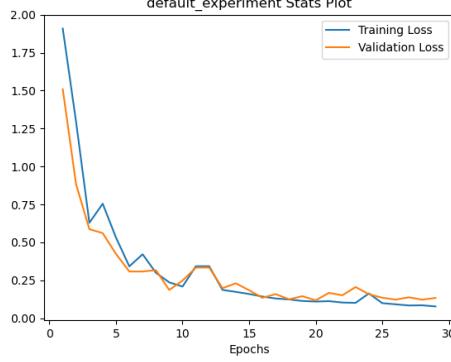


Figure 15: Losses Plot for Baseline ResNet-Transformer

We implemented the image position encoding as they did in [7]: half of features to encode the horizontal spatial information and half of the features to encode the vertical spatial information. The hyperparameters we used in the baseline model was shown below:

Table 1: Hyperparamter for Modified ResNet-Transformer

Learning Rate	Batch Size	Embedding Size	Feed Forward Size	# Decoder Layers	# Heads
5×10^{-5}	32	300	2048	8	4

The model performed well on test sets. One example, actual code and correctly rendered formula are:

$$T_{\{F\}}(z)=i\{\frac{2}{\alpha^{\prime}}\}\psi^{\mu}\partial X_{\mu}.$$

$$T_F(z) = i \frac{2}{\alpha'} \psi^\mu \partial X_\mu.$$

The predicted code and rendered formula are:

$$T_{\{F\}}(z)=i\frac{2}{\alpha^{\prime}}\psi^{\mu}\partial X_{\mu}.$$

$$T_F(z) = i \frac{2}{\alpha'} \psi^\mu \partial X_\mu.$$

Our predicted code rendered the same as the original code did, and it even observed a simplification, note a extra pair $\{\}$ is removed!

We observe one weakness of the model is dealing with the synonyms. For example the model cannot distinguish between the follow to symbols:

\rightarrow , \rightarrow

\rightarrow , \rightarrow

The second weakness is that the model does not behave well on the image outside the test set. It is because the images in our training and test sets have almost the same size. So for a random picture of formulas, we have to resize it to the same shape.

5.2.2 Best Run (without data augmentation) Model

To improve the performance of the ResNet-Transformer architecture, we proposed a minor change to the position encoding, we make two copies of the output from convolutions. Then add the horizontal encoding to one copy:

$$PE(y, 2i) = \sin \left(y/10000^{2i/d_{\text{model}}} \right)$$

$$PE(y, 2i + 1) = \cos \left(y/10000^{2i/d_{\text{model}}} \right)$$

and also add the vertical encoding to the other copy:

$$PE(x, d_{\text{model}}/2 + 2i) = \sin \left(x/10000^{2i/d_{\text{model}}} \right)$$

$$PE(x, d_{\text{model}}/2 + 2i + 1) = \cos \left(x/10000^{2i/d_{\text{model}}} \right)$$

Finally we concatenate the two results before we flatten them. Remember that we used 300 embedded dimension in the baseline model, but to do above position encoding, we'll have to use embedding size $d_{\text{model}} = 512$. So we essentially cut down the output of the bottle neck 1×1 convolution layer from 300 to 256. And the doubling feature position encoding will double channels to 512. The baseline model would have 8 layers in the transformer decoder part, but we cut down the decoder layers to 3 to reduce training time. The other hyperparameters are the same as the baseline, we listed in below table:

Table 2: Hyperparamter for Modified ResNet-Transformer

Learning Rate	Batch Size	Embedding Size	Feed Forward Size	# Decoder Layers	# Heads
5×10^{-5}	64	512	2048	3	4

The training loss plot is:

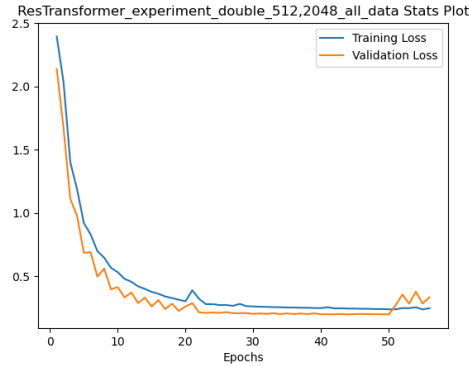


Figure 16: Losses Plot for Modified ResNet-Transformer

And we're able to get the BLEU-1 90.4% and BLEU-4: 77.7%.

5.2.3 Visualization

We visualize the gradient map of the best model using 'captum' module, the method we use is the integrated gradient visualization introduced in [8]. We find the model will look at mostly the correct positions as we would expect: The \LaTeX code begins with:

`\frac { \partial V } { \partial t^I } { \partial t^I } = \dots\dots`

$$\frac{\partial V}{\partial t^I} = \frac{\partial}{\partial t^I} C_{LMN} t^L t^M t^N = 3t_I.$$

Looking at the beginning of sentence

$$\frac{\partial V}{\partial t^I} = \frac{\partial}{\partial t^I} C_{LMN} t^L t^M t^N = 3t_I.$$

Looking at the fraction command

$$\frac{\partial V}{\partial t^I} = \frac{\partial}{\partial t^I} C_{LMN} t^L t^M t^N = 3t_I.$$

Looking at the first partial

$$\frac{\partial V}{\partial t^I} = \frac{\partial}{\partial t^I} C_{LMN} t^L t^M t^N = 3t_I.$$

Looking at the t

$$\frac{\partial V}{\partial t^I} = \frac{\partial}{\partial t^I} C_{LMN} t^L t^M t^N = 3t_I.$$

Looking at the first super script

But in many cases, the model would prefer look a little **ahead** as we will, an example, The \LaTeX code begins with:

`\varepsilon = \frac{ | c | ^ { 2 } } { 2 } \dots\dots`

$$\epsilon = \frac{|c|^2 (1 + r^2) - |a|^2}{2\sqrt{1 + r^2} |ab^* + bc^* (1 + r^2)|}$$

Looking at the beginning of sentence

$$\epsilon = \frac{|c|^2 (1 + r^2) - |a|^2}{2\sqrt{1 + r^2} |ab^* + bc^* (1 + r^2)|}$$

Looking at the ϵ

$$\epsilon = \frac{|c|^2 (1 + r^2) - |a|^2}{2\sqrt{1 + r^2} |ab^* + bc^* (1 + r^2)|}$$

Looking at the =

$$\epsilon = \frac{|c|^2 (1 + r^2) - |a|^2}{2\sqrt{1 + r^2} |ab^* + bc^* (1 + r^2)|}$$

Looking at the fraction command

$$\epsilon = \frac{|c|^2 (1 + r^2) - |a|^2}{2\sqrt{1 + r^2} |ab^* + bc^* (1 + r^2)|}$$

Looking at the left |

5.3 ViT

We faced a lot of memory issues while running the ViT model, largely because of the parallelization of transformers. For our original 128×1088 image, we had $(128 \times 1088)/(16 \times 16) = 544$ patches. Since all of these patches are processed parallelly, we always ran into CUDA_OUT_OF_MEMORY errors. After much thought, we decided on using 25% lower resolution images, and used $64 \times 544/(16 \times 16) = 136$ patches for the training. This led to us having about a 10 mins/epoch runtime.

We obtained a BLUE1 score of 91.53%, and a BLUE4 score of 77.06% on the test dataset. This is comparable to our best performing model (ResNet-Transformer - 93%).

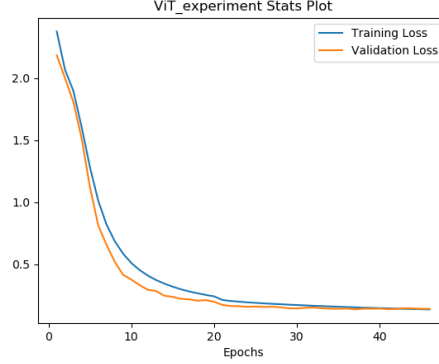


Figure 17: Losses Plot for Vision Transformer (ViT)

Table 3: Hyperparamters for ViT Encoder

Image Size	Learning Rate	Batch Size	Embedding Size	Feed Forward Size	# Decoder Layers	# Heads
96×816	5×10^{-5}	32	512	2048	8	8

We present some good examples and bad examples made by ViT model below:

$$\delta A_i = \theta \nabla^{-2} \epsilon^{ijk} \partial_j \pi_k, \quad \delta \pi_i = \theta \epsilon^{ijk} \partial_j A_k,$$

$$\delta A_i = \theta \nabla^{-2} \epsilon^{ijk} \partial_j \pi_k, \quad \delta \pi_i = \theta \epsilon^{ijk} \partial_j A_k,$$

Example 1

$$|m_2^2 - m_1^2| = (0.22 \pm 0.02) (eV/c^2)^2,$$

$$|m_2^2 - m_1^2| = (0.22 \pm 0.02) (eV/c^2)^2,$$

Example 2

$$J^a(z) = \phi(J_{-1}^a \Omega \otimes \overline{\Omega}; z, \bar{z}),$$

$$J^a(z) = \phi(J_{-1}^a \Omega \otimes \overline{\Omega}; z, \bar{z}),$$

Example 3

Figure 18: Good results with perfect match (Top actual, bottom predicted)

$$\psi_1\otimes\psi_2=\sum_i\psi_1^i\otimes\psi_2^i+\Delta_{1,0}(\mathcal{A}_{n+1})\left(\mathcal{H}_1\otimes\mathcal{H}_2\right)$$

$$\psi_2=\sum_i\psi_1^i\otimes\psi_2^i+\Delta_{1,0}(\mathcal{A}_{n+1})\left(\mathcal{H}_1\otimes\psi\right)$$

Example 4

$$\alpha_{GRR}(\mu^2)=\alpha_S(\mu^2)\left[1+\frac{\alpha_S(\mu^2)}{12\pi}\left(49-\frac{10n_f}{3}\right)\right]+\cdots$$

$$\langle P^2\rangle=\alpha_S(\mu^2)\left|1+\frac{\alpha_S g(\mu^*)}{12\pi}\left(49-\frac{10m_f}{3}\right)\right|$$

Example 5

$$\Delta\Pi=\frac{\alpha_s(Q^2)}{8\pi^3}\cdot\frac{1}{t_p}\cong\frac{b_0\alpha_s^2(Q^2)}{32\pi^4}$$

$$\Delta\Pi=\frac{\alpha_s(Q^2)}{8\pi^3\cdot\frac{1}{t_m}}\cong\frac{t_0\alpha_s^2(Q^2)}{32\pi^4}$$

Example 6

Figure 19: Results with some symbols missing/misplaced (Top actual, bottom predicted)

$$T_V(s)=\frac{\frac{RLs}{R+Ls}}{R+\frac{RLs}{R+Ls}}$$

$$\frac{\{d^{\wedge 2}\}\{dT^{\wedge 2}\}}{\left(\frac{1}{5+\sum_{d=1}^{\infty}n_{-d}d^{\wedge 3}}\right)^{\frac{e^{\wedge 2}}{(1-e^{\wedge 2})^{\wedge 2}}}\{dT^{\wedge 2}\}}J_{-i}=0$$

Example 8

$$\int_1^x\sum_{p\leq u}\left[\frac{\log u}{\log p}\right]\log p\,du=\frac{1}{2\pi i}\int_{c-i\infty}^{c+i\infty}\frac{x^{s+1}}{s(s+1)}\left(-\frac{\zeta'(s)}{\zeta(s)}\right)ds$$

$$\int_0^x\sum_{r\leq n}\left[\frac{\log u}{\log p}\right]\log p\,d\,ar=\frac{1}{2\pi}\int_{r=r+\infty+\infty}^{\frac{x^{\nu+1}}{s(s+1)}}\left(\frac{\zeta'(s)}{\zeta(s)\log}\right)$$

Example 9

Figure 20: Poor results (Top/left actual, bottom/right predicted)

5.4 ResNet with Global Context - Transformer

Post training our model, we achieved a loss of 0.15, a BLEU-1 score of 83.5% and a BLEU-14 score of 73.1% on the test dataset. Since, the architecture is not inherently parallel (like ViT), neither were we able to fully train the model, nor were able to tune the hyperparameters. The model took about 20 minutes per epoch to train, which constrained our computational resources and hindered us from achieving similar results in [5]. We adopted a mix of parameters based on experience from the already tested models (ViT and ResNet-Transformer). Nonetheless, we were able to achieve respectable results, and believe that these can be very easily improved with additional resources and time. The plot for loss on training and validation dataset with respect to the number of epochs can be seen in Fig 21.

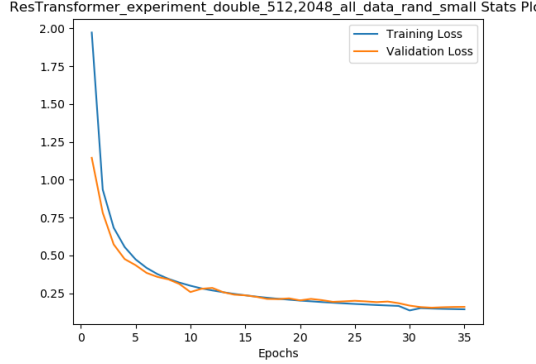


Figure 21: Losses Plot for ResNet with Global Context

The hyperparameters used for training can be seen in 4. Overall, the model performed well on most of the examples. There were some rare cases where the model was not able to produced a $\text{L}^{\text{T}}\text{E}^{\text{X}}$ code, which could be rendered into a formula.

Table 4: Hyperparamters for ResNet with Global Context

Learning Rate	Batch Size	Embedding Size	Feed Forward Size	# Decoder Layers	# Heads
5×10^{-5}	32	512	2048	3	4

One such test case, where the model misses out on closing a brace and a digit too is as follows:

The actual code and correctly rendered formula are:

$$25 \, eV \sim \{2\} \stackrel{\text{rel}}{\sim} \{<\} \{ \sim \} | \, \delta m_{\tau s} \sim \{2\} | \stackrel{\text{rel}}{\sim} \{<\} \{ \sim \} 100 \, eV \sim \{2\} ,$$

$$25 \, eV^2 \lesssim |\delta m_{\tau s}^2| \lesssim 100 \, eV^2 ,$$

The predicted code is:

$$25 \sim eV \sim \{2\} \stackrel{\text{rel}}{\sim} \{<\} \{ \sim \} | \, \delta m_{\tau s} \sim \{2\} | \stackrel{\text{rel}}{\sim} \{<\} \{ \sim \} \{10 \sim eV \sim \{2\} ,$$

Another test case where the formula is correctly rendered is:

$$\{ \mathcal{L} \} = \{ \frac{1}{2} \} \{ 2 \} \} \partial_{\mu} \phi \partial^{\mu} \phi - \{ \frac{1}{2} \} m^2 \phi^2 - \frac{g}{3!} \phi^3 ,$$

$$\mathcal{L} = \frac{1}{2} \partial_{\mu} \phi \partial^{\mu} \phi - \frac{1}{2} m^2 \phi^2 - \frac{g}{3!} \phi^3 ,$$

The predicted caption and formula is:

$$\begin{aligned} \{ \text{\cal L} \} = & \{ \frac{1}{2} \} \{ 2 \} \} \partial_{\mu} \{ \text{\mu} \} \\ & \phi \partial^{\mu} \phi - \{ \frac{1}{2} \} \{ 2 \} \} \\ & m^2 \phi^2 - \{ \frac{g}{3!} \} \{ 3 \} \} \\ & \phi^3, \end{aligned}$$

$$\mathcal{L} = \frac{1}{2} \partial_{\mu} \phi \partial^{\mu} \phi - \frac{1}{2} m^2 \phi^2 - \frac{g}{3!} \phi^3,$$

The following is a peculiar test case, where the model predicted a grammatically correct formula; however, added a few extra elements.

$$\begin{aligned} \alpha_{\lambda,2} = & \frac{\alpha_{\lambda_0}}{1 + \frac{\alpha_{\lambda_0}}{2\pi} \ln \left(\frac{\lambda}{\lambda_0} \right)} . \\ & \{ 0 \} \} \{ 1 + \frac{\alpha_{\lambda_0}}{2\pi} \ln \left(\frac{\lambda}{\lambda_0} \right) \} \{ 2 \pi \} \; ; \; \mathrm{ln} \left(\frac{\lambda}{\lambda_0} \right) \; ; \; . \end{aligned}$$

$$\alpha_{\lambda,2} = \frac{\alpha_{\lambda_0}}{1 + \frac{\alpha_{\lambda_0}}{2\pi} \ln \left(\frac{\lambda}{\lambda_0} \right)} .$$

The predicted caption and formula is:

$$\begin{aligned} \begin{array}{l} \{ \alpha_{\lambda,2} = \frac{\alpha_{\lambda_0}}{1 + \frac{\alpha_{\lambda_0}}{2\pi} \ln \left(\frac{\lambda}{\lambda_0} \right)} . \\ \{ 0 \} \} \{ 1 + \frac{\alpha_{\lambda_0}}{2\pi} \ln \left(\frac{\lambda}{\lambda_0} \right) \} \{ 2 \pi \} \} \\ \backslash, \operatorname{ln} \left(\frac{\lambda}{\lambda_0} \right) \} \backslash \end{array} \end{aligned}$$

$$\alpha_{\lambda,2} = \frac{\alpha_{\lambda_0}}{1 + \frac{\alpha_{\lambda_0}}{2\pi} \ln \left(\frac{\lambda}{\lambda_0} \right)}$$

5.5 Generalization of ResNet-Transformer

5.5.1 Baseline Random Scaling

As discussed in the Methods section, to make our model really work in practice, we need to make robust to size changes, so we use the same hyperparameter as the best run ResNet-Transformer model:

Table 5: Hyperparamters

Learning Rate	Batch Size	Embedding Size	Feed Forward Size	# Decoder Layers	# Heads
5×10^{-5}	64	512	2048	3	4

And we impose a random scaling from 0.8 to 1.4 to all the images. Then we retrain the model, the loss curves are as follows:

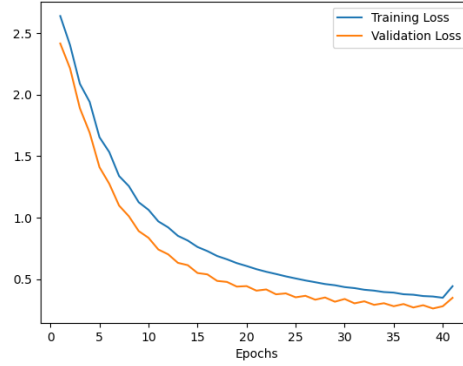


Figure 22: Losses Plot for ResNet-Transformer(Baseline) with Random Scale

And we are able to recognize images obtained from screenshot images, see the following examples:

$$\Omega_{K3} := \{[\sigma] \in \mathbb{P}(\Lambda_{K3} \otimes \mathbb{C}) \mid \langle \sigma, \sigma \rangle = 0, \langle \sigma, \bar{\sigma} \rangle > 0\}.$$

Screen shot image, not an image from dataset!

The model predicts:

$$\Omega_{K3} := \{ [\sigma] \in \mathbb{P}(\Lambda_{K3} \otimes \mathbb{C}) \mid \langle \sigma, \sigma \rangle = 0, \langle \sigma, \bar{\sigma} \rangle > 0 \}$$

which unfortunately cannot render due to unmatched braces, but it get mostly the correct code!

The BLEU-1 score ends up at 86.9% and BLEU-4: 72.7%.

5.5.2 Random Scaling and Salt-Pepper Noise

Though the ResNet-Transformer trained with random scaling can deal with screen-shots, but it's sensitive with background cleanness. Though we assume in most of the use cases, the background are purely white, but there can be extra pixels appears in the boundary. So we tried to add the data augmentation that randomly turn a white pixel black with probability 2%.

The loss curves are as follows, we observe the losses are higher than the no noised version:

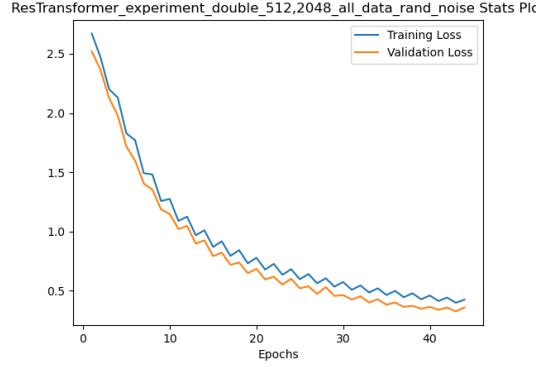


Figure 23: Losses Plot for ResNet-Transformer(Baseline) with Random Scale and Noise

It turns out the BLEU-1 score is only 75.4% and BLEU-4 score 59.5%, which indicate the model is not practically usable, so we give up the salt-pepper noise approach and try to tune hyperparameter more carefully instead.

5.5.3 Best Run Random Scaling

The baseline model with random scaling seems already promising, we did a further hyperparameter search, and it turns out double the number of transformer decoder layers and half the batch size would lead to a performance surge. The best hyperparameters we found are:

Table 6: Best Run Hyperparamters

Learning Rate	Batch Size	Embedding Size	Feed Forward Size	# Decoder Layers	# Heads
5×10^{-5}	32	512	2048	6	4

The loss curves are as follows:

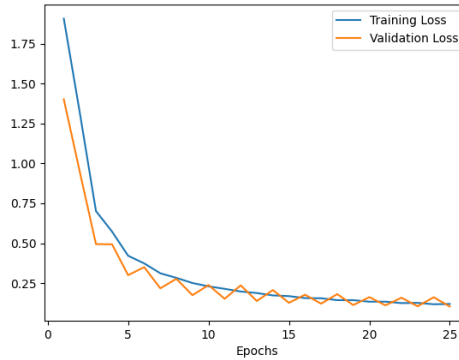


Figure 24: Losses Plot for ResNet-Transformer(Best) with Random Scale

The losses are all lower than the ones we have for baseline model, and for the BLEU-1 score, we get 93.3% and BLEU-4 83.8%, even better than the ones without random scaling!

The model is robust enough to handle complicated one-line equation, below are some examples:

$$Z_{GW,\beta} = \sum_{g=0}^{\infty} r_{g,\beta} u^{2g-2} \left(\frac{\sin(u/2)}{u/2} \right)^{2g-2}.$$

Screen shot image, not an image from dataset!

The predicted code is long, we only present the reconstructed image:

$$Z_{GW,\beta} = \sum_{g=0}^{\infty} r_{g,\beta} u^{2g-2} \left(\frac{\sin(u/2)}{u/2} \right)^{2g-2}$$

A more complicated example:

$$\sum_{g=0}^{\infty} \sum_{h=0}^{\infty} R_{g,h} u^{2g-2} q^{h-1} = \frac{1}{u^2 \Delta(q)} \cdot \exp \left(\sum_{g=1}^{\infty} u^{2g} \frac{|B_{2g}|}{g \cdot (2g)!} E_{2g}(q) \right).$$

Screen shot image, not an image from dataset!

And the model can reconstruct the follows, we can see it missed one summation sign, but other details are recovered correctly:

$$\sum_{g=0}^{\infty} R_{g,h} u^{2g-2} q^{h-1} = \frac{1}{u^2 \Delta(q)} \cdot \exp \left(\sum_{g=1}^{\infty} u^{2g} \frac{|B_{2g}|}{g \cdot (2g)!} E_{2g}(q) \right).$$

However the model is not perfect, it cannot handle multi-line formulas due to the selection we made for the image size. Even the image is perfectly rendered one line easy equation, the model will sometimes produce the repetition, just like in the CALSTM, below is one example:

$$R^i(\mathcal{M}_g) \times R^{g-2-i}(\mathcal{M}_g) \rightarrow R^{g-2}(\mathcal{M}_g)$$

Screen shot image, not an image from dataset!

And the model gives code which rendered as:

$$R^i(\mathcal{M}_g) \times R^{g-2-i}(\mathcal{M}_g) \rightarrow R^{g-2}(\mathcal{M}_g) \rightarrow R^{g-2}(\mathcal{M}_g) \cdot \rho \cdot \rho$$

We can see the last term in the sequence was repeated twice and the model cannot stop generation properly as well.

5.6 Beam Search

We used beam search in place of greedy search for the best CALSTM and the ResNet-Transformer models in the predict phase of the testing scheme in order to get the BLEU scores. For the CALSTM model, we achieved 77.43 BLEU-1 and 56.81 BLEU-4 with beam search. For the ResNet-Transformer best model, we achieved 93.34 BLEU 1 and 83.81 BLEU 4 with a beam width of 10. We increased the beam width to 30 in order to test a wider exploratory field and received indistinguishably better answers for both models; they were identical within 0.50% for both BLEU scores.

6 Discussion

6.1 CALSTM

Our CALSTM implementation did not perform as well as the original in [6], most likely due to our limitations for training only 30 epochs, as opposed to the original's 100. Additionally, the CALSTM model performed the worst of our four models, due to its much simpler attention mechanism, as it was only able to give attention to a small collection of receptive fields at a given time. The CALSTM's recurrent nature also made it our most computationally expensive model to train, with a single epoch taking approximately 20 minutes.

6.2 ResNet-Transformer and Generalization

We find the simplest architecture works the best for the task. We did a tweak on the 2D position encoding which boost the performance. The architecture is robust enough to generalize, we achieved the highest performance with random scaling data augmentation. To train this type of model, we find choosing the correct batch size is the key, a batch size 64 training will never outperform a batch size 32 even with more epochs.

We use integrated gradient method to visualize the model, we find its attention would most aligned with human intuition, but sometimes would move ahead from the desired positions. We believe it's attention mechanism that causing we have repetition issues when making predictions, but we didn't get a good fix for that.

6.3 ViT

We expected the Vision Transformer based model to perform better than the ResNet model, as it is designed to model the spatial and sequential dependencies better than the existing Convolutional Neural Networks. Moreover, its implementation is much faster than the ResNet model with similar number of trainable parameters, but it did not allow us to scale up the image size, which did hinder the modeling of peculiar characters.

We see that for Figures in 18, we get a perfect match in the actual and predicted formulas. For Figures in 19, we get most of the formula right, but some occasional slip-ups in the symbols. For Figures 20, we get completely wrong (or even non-render-able) outputs. This is because sometimes the curly braces $\{ \}$ are misplaced, which means there is either a syntax error or the equation renders in the superscript or subscript.

6.4 ResNet with Global Context - Transformer

Similarly, this network was also expected to perform better than the simple ResNet-Transformer based network; however, we could not allow it to train fully. With sub-optimal hyperparameters, we had to artificially create an early-stop epoch. As expected, it resulted in a sub-optimal results. Overall, we deduced that the time and memory complexity involved with this model is not worth the results, and the future belongs to purely transformer based networks.

The results indicate that the model is not able to pay attention to repeated elements, which are closely spaced in the input image. We have experienced certain cases, where the network is not able to predict array elements correctly or miss repeated subscripts too.

6.5 Beam Search

Beam search neither helped nor hindered the accuracy of the LaTeX predictive outputs. For the best models in the CALSTM and ResNet-Transformer architectures, the use of beam search in place of greedy search produced indistinguishable results from the best greedy LaTeX captions, despite the breadth of beams we searched. When the greedy and the beam search predict with roughly the same accuracy to the actual caption, this means that the best models have achieved the limit of their predictive capabilities given the hyper-parameters and number training epochs. It is worth confirming this by testing beam search on a less-well-trained model; for example, one that only ran for 15 or 20 epochs. Then it is possible to see how beam search can make training these models more efficient by

achieving the same BLEU score with fewer epochs of training than greedy would produce with an asymptotically large number of epochs.

Additionally, with more time, beam search can be parameterized to give penalties to captions that, for example, repeat the same series of symbols in a loop which negatively impacts the BLEU score. We could also give beam search a temperature and allow it to explore more random values to journey down less-likely paths that might end up being more probable in later caption positions.

7 Limitations and Further Directions

From testing the models on real formula images, we find the following limitations:

1. The models cannot handle large multi-line equations.
2. The models will have repetition problems when dealing with formulas which contains similar sub-parts.
3. The models cannot recognize formula images which contains any significant noise.
4. The model usually fail to close a pair of braces correctly, and thus cause syntax errors.

The first limitation coming from the construction of the training/validation/testing set, we deliberately deleted the large formulas with height greater than 128 pixels. If we trained on all the formula images given by the dataset author, this problem can be resolved in the future.

For the second problem, we suspect it comes from the attention mechanism, we haven't have an idea to resolve that yet. It's worth to investigate more on this issue in the future.

For the third problem, we tried to resolve it by data augmentation but the results are not good. We believe a smart way of doing preprocess can solve it, and the preprocess should be possible within the scope of traditional vision methods.

Lastly for syntax error, we observe this issue doesn't happen to the commercial software "Mathpix", and we suspect they might have some extra terms in the loss function that penalize the syntax errors. But we're out of time to investigate that.

8 Authors' Contributions

1. **Cameron Cinel:** I worked on the PyTorch implementation of the CALSTM model. I also created the methods to give a visualization of the attention mechanism in the model. Additionally, I wrote the methods, results, and discussion sections for the CALSTM model.
2. **Bochao Kong:** I implemented the data loading for the dataset Image2Latex 140K (adapted from PA4), I implemented part of the ResNet-Transformer model and worked out the data augmentation for generalization to real-life formula images. I made the visualization of the ResNet-Transformer model and a demo that can be used to translated screen-shot image to Latex code.

For the report, I write the abstract, introduction, most parts related to the ResNet-Transformer model.

3. **Yuzhe Bai:** I wrote the code for loading configuration and saving experiment data and models. Also, I helped writing of the encoder part of the ResNet-Transformer model as well as image preprocessing. I also implanted noise tranformation and ran several experiments. For the report, I wrote the part of dataloading and preprocessing.

4. **Shrey Kansal:**

I worked on setting up the Vision Transformer model, and adapting it to our use case of predicting sequential outputs, instead of classifying the objects in an image. I also contributed for hyperparameter tuning and generating different results Furthermore, I worked on developing the ResNet with Global Context based model, which involved restructuring the encoder and introduced a few extra hyperparameters. I got it working with both ResNet18 and ResNet34; however, I couldn't implement the latter to full extent due to computational constraints. For the report, I contributed to the aforementioned two models and produced results for discussion.

5. **Rishabh Bhattacharya:** I worked on the Vision Transformer model, which involved research, including reading up papers, understand a few implementations, and finally modify them according to our project. I spent a lot of time debugging, testing, and tuning hyperparameters to arrive at the right model, and were finally able to get the network running on 25% resolution images. I also worked on the report section for ViT, and generating the test results for the same. Finally, I helped in organising the code repository to make it more readable.
6. **Madeleine Kerr:** I worked on designing, implementing, debugging, and testing the beam search. I read several papers on this topic and I spent most of the time debugging and configuring the algorithm to meet the need and structure of the CALSTM and ResNet-Transformer models we tested it on, to no improvement (but to no detriment either, so...). The debugging took a very long time since I'm a relative PyTorch newbie. I worked on the beam search parts of the report and on merging these multiple versions of the code with beam search into an integrated final version.
7. **Zichen He:** I implemented part of the ResNet-Transformer model and found one critical bug about loss functions. I ran experiments on the baseline model. Additionally, I wrote the methods, results sections for the baseline model of ResNet-Transformer.

References

- [1] Robert H Anderson. Syntax-directed recognition of hand-printed two-dimensional mathematics. In *Symposium on interactive systems for experimental applied mathematics: Proceedings of the Association for Computing Machinery Inc. Symposium*, pages 436–459, 1967.
- [2] Daniela S Costa, Carlos AB Mello, and Marcelo d’Amorim. A comparative study on methods and tools for handwritten mathematical expression recognition. In *Proceedings of the 21st ACM Symposium on Document Engineering*, pages 1–4, 2021.
- [3] Yuntian Deng, Anssi Kanervisto, Jeffrey Ling, and Alexander M Rush. Image-to-markup generation with coarse-to-fine attention. In *International Conference on Machine Learning*, pages 980–989. PMLR, 2017.
- [4] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.
- [5] Nuo Pang, Chun Yang, Xiaobin Zhu, Jixuan Li, and Xu-Cheng Yin. Global context-based network with transformer for image2latex. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 4650–4656, 2021.
- [6] Sumeet S. Singh. Teaching machines to code: Neural markup generation with visual attention, 2018. <https://arxiv.org/abs/1802.05415>.
- [7] Sumeet S Singh and Sergey Karayev. Full page handwriting recognition via image to sequence extraction. In *Document Analysis and Recognition-ICDAR 2021: 16th International Conference, Lausanne, Switzerland, September 5–10, 2021, Proceedings, Part III 16*, pages 55–69. Springer, 2021.
- [8] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *International conference on machine learning*, pages 3319–3328. PMLR, 2017.
- [9] Masakazu Suzuki, Fumikazu Tamari, Ryoji Fukuda, Seiichi Uchida, and Toshihiro Kanahori. Infty: an integrated ocr system for mathematical documents. In *Proceedings of the 2003 ACM symposium on Document engineering*, pages 95–104, 2003.
- [10] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.