

Deep Learning Project

1. Overview

This project focuses on the task of converting an image of an equation into its corresponding LaTeX formula, inspired by the Image-to-Markup project created and published (*arXiv:1609.04938v2*) by HarvardNLP. Although we experimented with other approaches, we eventually settled on an encoder-decoder model which the original research authors used as well. Note that due to the time and resources available to us, we only focused on processing equations that contained 30 tokens or less.

2. Dataset

The team originally intended to use the IM2LATEX-100k dataset provided by the original paper, containing 103,556 formulas and their corresponding images of fixed A4 sizes. However, there was a lot of white space in the images as the formulae only took up 1-2 lines. This was a big source of inefficiency in training our model, especially since we need to perform computationally heavy operations for the task.

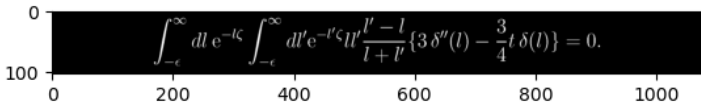
Through more research, we found a cleaned dataset by [source here]. The images were processed such that the image size was the size of the formulae (non-uniform size across the dataset). LaTeX tokens that render as the same/similar outputs are standardised to reduce the variation in the dictionary. The corresponding LaTeX formulas were also normalised by having a "\space" character between each token to ensure clarity.

2.1 Data engineering

An inversion of colours of the images such that their background colours were black. Due to the non-uniform size of images (and corresponding formulae) across the dataset, we encountered difficulties in batching training samples. This was overcome by writing a custom `collate_fn` for the torch data loader which:

1. Padded images to the largest image size in its batch.
2. Padded tokenized formulae to the longest length in its batch.

We provide an example:

Processed Image	
------------------------	--------------------------------------------------------------------------------------

Padded Formula	$\int_{-\epsilon}^{\infty} dI: \mathrm{e}^{-I\zeta} \int_{-\epsilon}^{\infty} dI^{\prime} \mathrm{e}^{-I^{\prime}\zeta} \left I^{\prime} \left\{ \frac{I^{\prime}}{I} - 1 \right\} + I^{\prime} \left\{ \frac{3}{4} \right\} \right \Delta(I) = 0.$
-----------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

2.2 Dataset Pruning

As the dataset is large and model training is computationally heavy, we only used a subset of the dataset with 30 or less tokens in the formulae. The train-test split is as shown below:

Data	Number of Samples
Train	12265
Test	616

Training over the entire dataset required approximately 50 hours per epoch. After pruning, we were able to reduce this to 1.5 hours per epoch.

3. Model Architecture

3.1 Overview

The original Harvard team used an encoder-decoder model based on a scalable coarse-to-fine attention mechanism, originally developed using the Torch library written in Lua.

Our project uses PyTorch and similarly uses an encoder-decoder model, although we have simplified it for the purposes of this project. Our encoder is made up of convolutional layers and a gated recurrent unit to encode a feature map of the input equation image, while our decoder is also made up of a gated recurrent unit with a predictive alignment attention mechanism to produce the corresponding LaTeX formulas.

Below is an illustration providing an overview of our model.

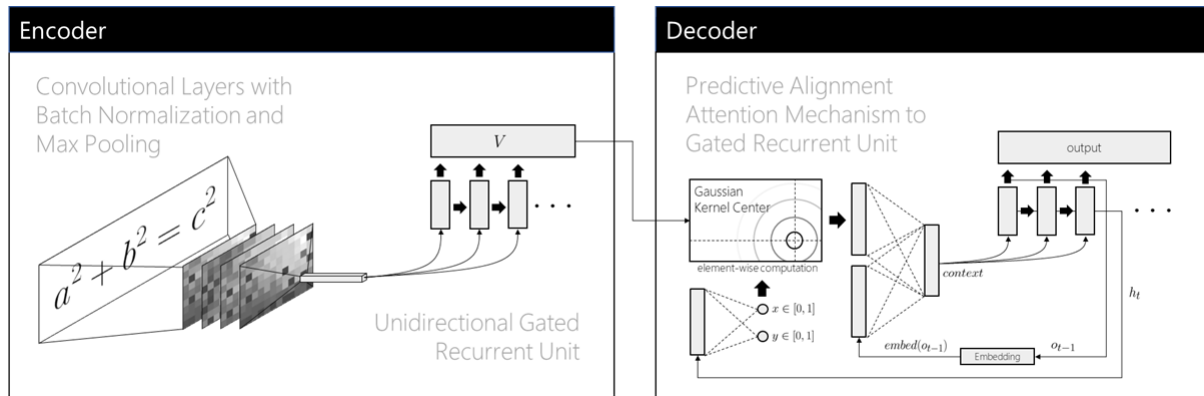


Figure 3.1a: Overview of Model

3.2 Encoder

The $1 \times H \times W$ input image is passed through a sequence of convolutional layers to obtain a $C \times H' \times W'$ feature map, which is then processed by a Gated Recurrent Unit (GRU). The GRU acts as a row encoder which encodes information in the feature map from left-to-right, similar to how an equation is read in reality by humans.

Research ([arXiv:1412.3555v1](#)) showed that GRUs are computationally more efficient and can perform comparably with LSTMs on smaller datasets, hence it was our RNN of choice given the scope and data available to us.

Every row within the feature map produced by the convolutional layers is run through the GRU, and the outputs are stacked to create a $D \times H' \times W'$ matrix, V .

3.3 Decoder

Like the encoder, the decoder is a GRU. The GRU is initialized with a start-of-sentence token as the first input, and the average of the V matrix across H' and W' as the first hidden state. Afterward, the output from the previous time step is used as the next time step's input.

At each time step, the input is first converted into an embedding. The embedding, together with the hidden state and the V matrix, are passed to a predictive alignment attention mechanism, the output of which serves as the actual input to the GRU cell. The cell output is then run through a linear layer that produces the classification probabilities for each of the output tokens.

The predictive alignment attention works by means of an alignment mechanism, which comprises two linear layers with Tanh and Sigmoid activation. The alignment mechanism takes in the hidden state as input, and outputs two numbers in the range of $[0, 1]$, which serve as coordinates to the image: one representing X and the other representing Y . The coordinates are projected onto the V matrix as the “center of attention”. A Gaussian kernel is computed around the center of attention, and is used as the weighting for each element in the V matrix. We then multiply the weighting with the V matrix to produce a single vector of dimension D . This, together with the input embedding, is run through a final linear layer to generate the output of the attention block.

Predictive alignment attention was selected due to its speed compared to global attention, and also because it is an intuitive choice for an image transcription problem - it makes sense that the decoder will want to sequentially focus on different parts of the image when generating each individual token (in other words, it is very rare for the decoder to need to look at two entirely different parts of the image in order to produce a single token). Modelling attention as a singular point of focus thus makes sense for this problem. In practice, we found that the attention block produced very sensible outputs: during decoding, the X coordinate generated by the alignment mechanism increased monotonically from 0 to 1, reflecting that the decoder was “scanning” through the image from left to right in order to generate outputs, mimicking human behaviour.

4. Training

4.1 Model Selection

The team tried several models in our implementation as described below:

1. **Encoding input images.** Autoencoders were used to learn a compressed representation of each image to be fed into the row encoder. However, this proved to be redundant and did not provide any improvement to model training. The convolutional layers used in the model as explained in Section 3.1 were adequate for the task.
2. **Formulae embeddings.** SkipGrams were trained on the normalised formulae dataset to obtain useful embeddings for training. This step was done as an initial method to gain semantic information between formulae tokens. However, the training for SkipGrams is computationally expensive - 1 epoch for ~ 50 million samples took 21 hours to compute. Unfortunately, the embeddings did not provide much improvement to the base model, and we decided not to spend more resources trying to train better embeddings by doing more epochs. This decision was made in conjunction with our choice to try attention layers, which could theoretically also capture context.
3. **Using global attention.** Our initial implementation of an attention mechanism ran each vector of dimension D within the V matrix through a set of linear layers to determine the weightage for that element. However, doing so was computationally heavy, as it required the linear layer to be computed $H \times W$ times. As we have limited time and computing resources, we decided instead to implement predictive alignment as explained in Section 3.3 above.
4. **Transformer.** We attempted a transformer architecture using default PyTorch parameters, with a convolutional layer functioning as an embedding for the input image. We found that this model was able to train and converge extremely quickly (about 7 minutes per epoch, and 5 epochs for convergence) due to its parallel nature,

as opposed to the sequential architecture of RNN-based models. However, its performance was not as good as the model which we finally selected; it had a test loss of about 1.65, compared to 1.25 for the selected model.

4.2 Hyperparameters Selection

Unfortunately, the high complexity of the model meant training required huge amounts of time and computing resources. Thus, the group was unable to conduct a full hyperparameter search. However, we tried our best in using empirically proven-to-work parameters from published state-of-the-art models.

4.3 Training Performance

The model was trained for a total of 52 epochs over the course of 5 days. Figure below shows our training and test loss curves.

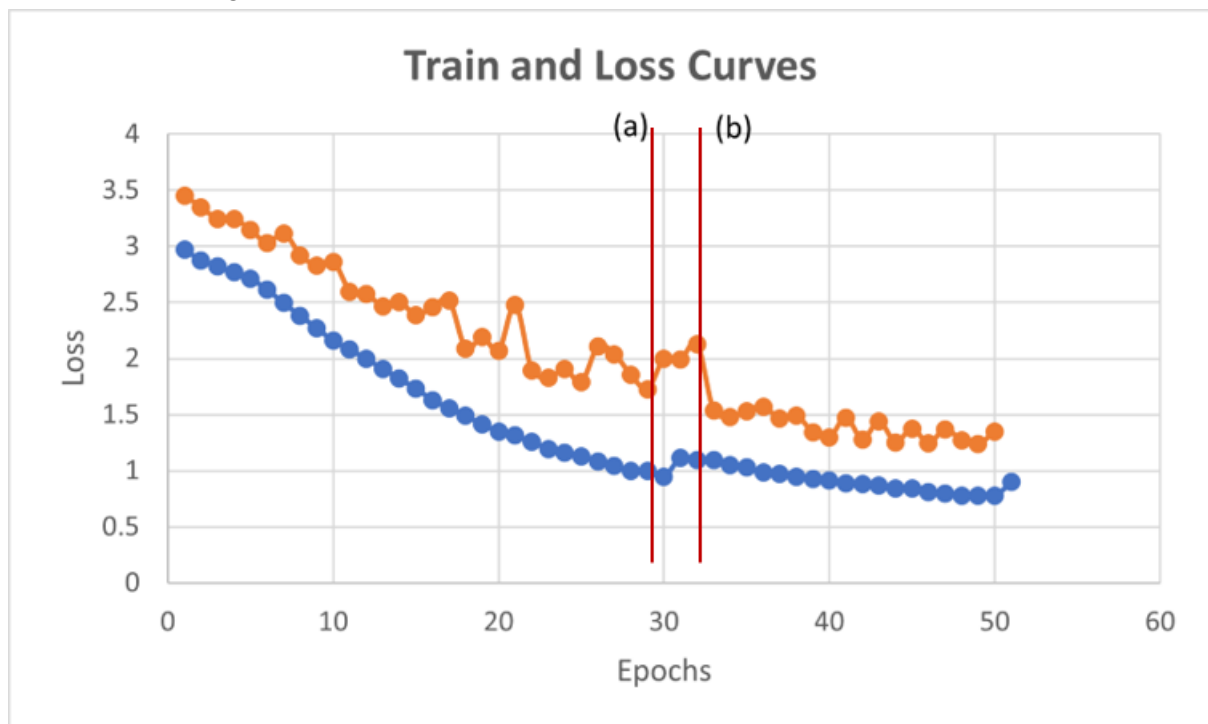


Figure 4.3a: Training and Test Loss Curve

We used teacher forcing, a technique where the correct output from the previous time step, rather than the predicted output from the model, is fed as input to the next cell, as it is known to result in faster convergence. Initially, training was done with 50% of the training samples being teacher forced. There is a spike in training loss between (a) and (b) as we reduced this ratio to 25% in order to encourage the model to be able to produce correct outputs independently. Decrease in validation loss is observed after the spikes, indicating that the model is training well.

Our team built a simple graphical user interface (GUI) for end-users to interact with easily (Figure 5a).

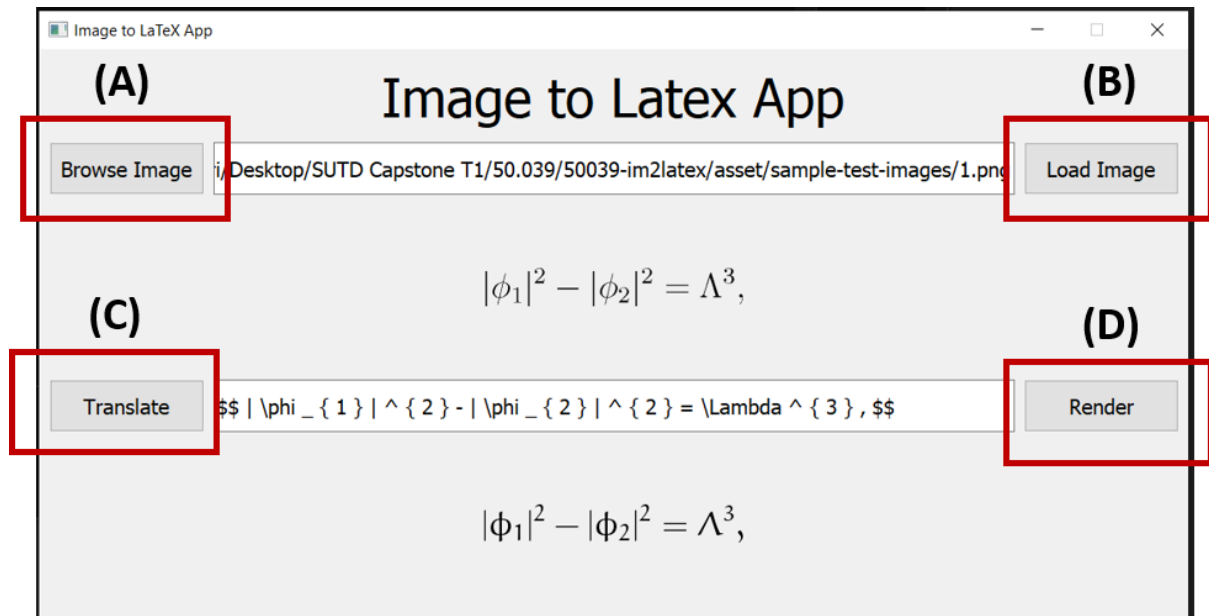


Figure 5a: GUI

5.1 Instructions

- A. Upon launching the app, the user will see an interface such as in Figure 5a.
- B. To upload an image of a formula to be converted, the user can click on (A) “Browse Image” to launch his local computer directory.
- C. Once the path of the input image is selected, click (B) “Load Image”.
- D. Then click (C) “Translate”. This step feeds the input image into our trained model, which outputs a sequence of formulae tokens. The predicted sequence can be seen in the white bar.
- E. Lastly, click (D) “Render” to obtain a LaTeX image of the output equation. This is for convenient checking of input image against output.

5.2 Some Examples

We provide some examples below:

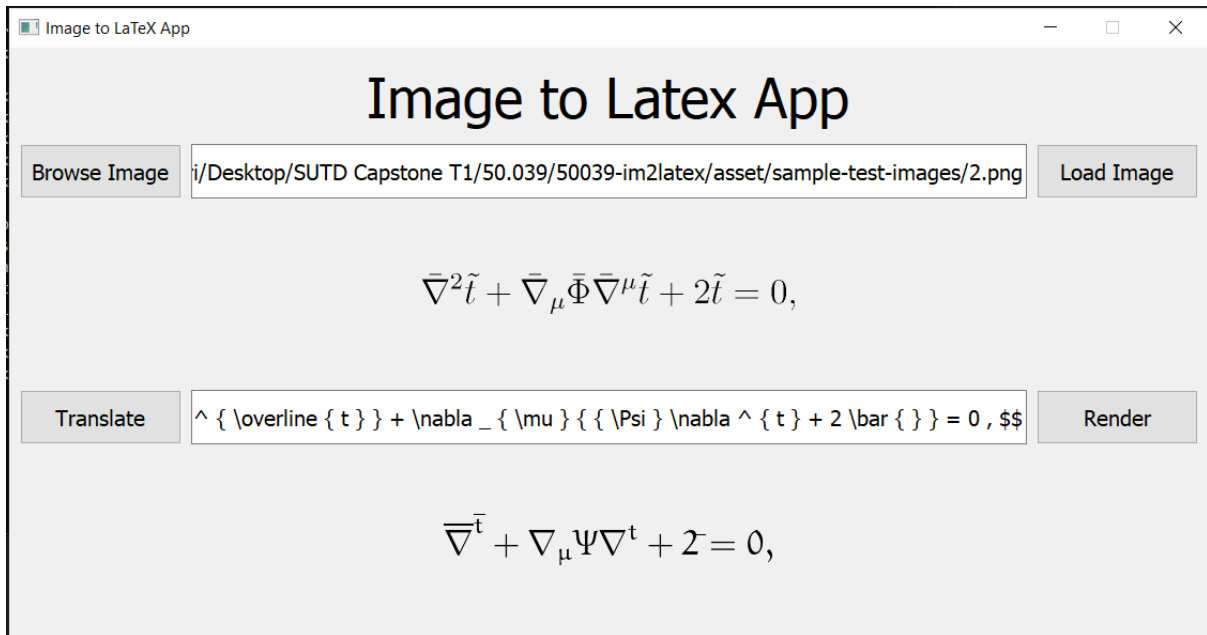


Figure 5.2a: Example 1

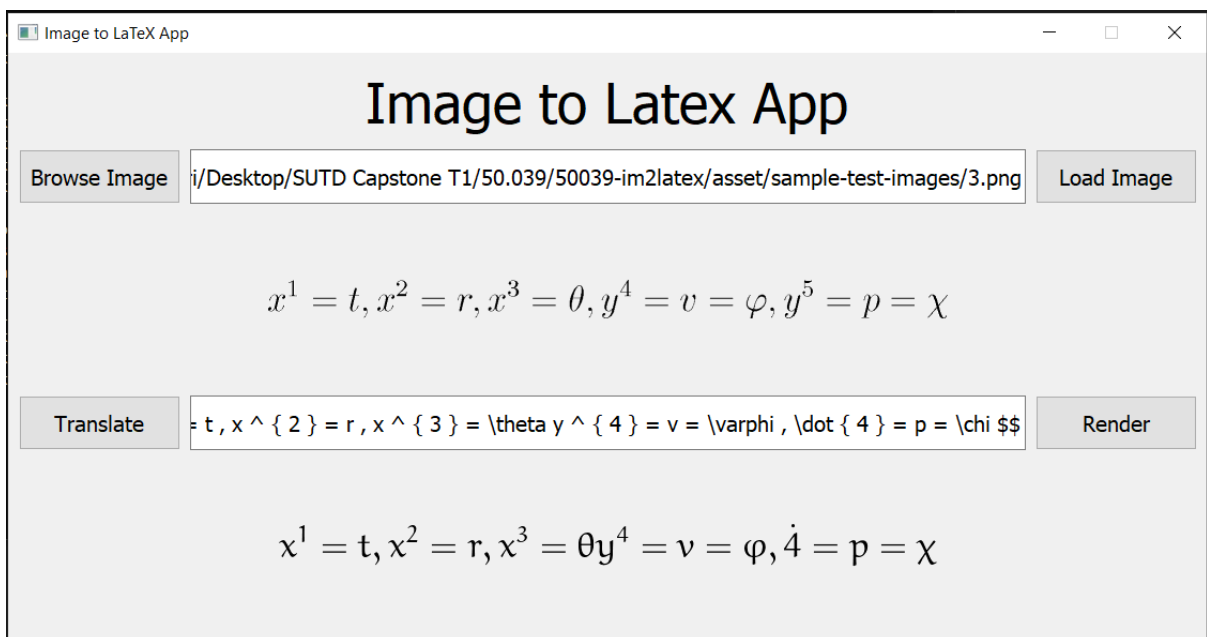


Figure 5.2b: Example 2

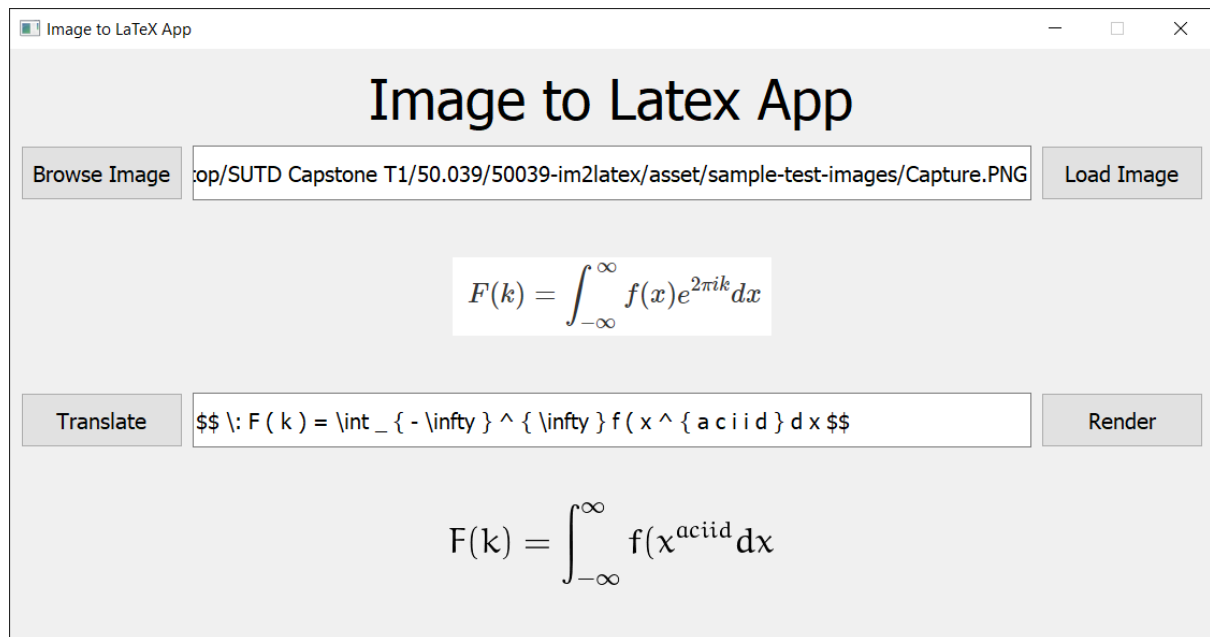


Figure 5.2c: Example 3

Overall, although the model is not 100% accurate, its performance is surprisingly adequate given the limited amount of time and computing resources. Referring to examples shown above, the model mostly struggles with superscripts in the formulae.

6. Future Work