Project Report : CS 4644

Alejandro Hinojosa Canada
acanada6@gatech.edu

Pablo San Francisco
pfrancisco6@gatech.edu

Yuxiang Qiu
yuxqiu@proton.me

Georgia Institute of Technology, North Ave NW, Atlanta, GA 30332, USA

Abstract

Over the past few years, text recognition has become an important task, and an important subset of this discipline is formula recognition, i.e., the translation of complex mathematical symbols from images into LaTeX code. While some solutions exist, they are either costly, require proprietary software, or do not utilize state-of-the-art architectures. This project aims to design a model that achieves state-of-the-art performance while allowing users to run the model locally. To achieve this goal, we implemented several state-of-the-art encoders and decoders and experimentally evaluated their performance, which was shown to be reasonably good. In this report, we present our approach, describe our experimental setup, analyze the results, and discuss future work to improve the model.

1. Introduction/Background/Motivation

The conversion of complex mathematical functions from images into LaTeX format is a critical tool for academic and technical documentation. Despite existing solutions, there remains a significant need for more accessible, efficient, and cost-effective methods. Current tools are not open-source and costly [1], limiting their improvement and adaptation by the academic community.

Our project, *LaTeXify*, aims to develop a local, free tool that is computationally inexpensive to convert images of mathematical functions into LaTeX code, directly addressing the shortcomings of current non-open-source, paid solutions, making educational materials more accessible, and facilitating the exchange of scientific ideas.

Specifically, our work involves building an encoder that extracts information from images and a decoder that outputs LaTeX code. This includes integrating and testing different CNN architectures such as ConvNeXt [2] and Swin Transformers [3], as well as different decoders such as Transformer [4] and GPT [5]. At the same time, our project seeks to contribute general architectural guidelines for designing image-to-text systems, which could be beneficial across various domains where image-based data need to be converted into editable text formats.

The field of transforming images into LaTeX code has seen various approaches, with significant contributions from the academic community aiming to improve accuracy and reduce computational overhead. Early methods employed traditional image processing techniques, but recent advances have leveraged deep learning for more precise results.

For instance, [4] introduced a global context-based network utilizing Transformer architecture to enhance the feature extraction process, achieving significant improvements in accuracy and achieving a final BLEU score of 89.72%. [6] explored the use of visual attention to facilitate image-to-LaTeX conversion, utilizing a model without feature pooling and achieveing a BLEU score of 89.09%. [7] applied a Vision Transformer (ViT) to solve this problem, highlighting the potential of Transformers to solve this problem and achieving a BLEU score of 88%.

2. Approach

Our model architecture consists of an encoder and a decoder, where the encoder extracts the feature representation from the input formula image, and then passes the features to the decoder, which combines these with the output embedding to predict the next possible token of each position through the attention mechanism.

The reason we think our model can beat the current SOTA in this task is because we use a SOTA encoder and a SOTA decoder. By combining the SOTA encoder and decoder, it is reasonable to expect that our model will outperform previous models based on ResNet [4] or vanilla CNN encoders [6, 8] and RNN [8], LSTM [6] or GRU [4] decoders.

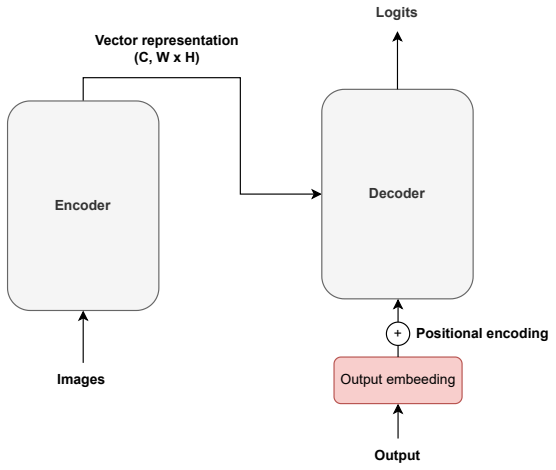

Figure 1. Architecture of *Latexify* model

2.1. Dataset

The dataset chosen for this task was *I2L-140k* [6]. This dataset consists of 140k images and is the end result of preprocessing the im2latex-100k dataset [9], which involves parsing the latex formulas, filtering and normalizing the extracted formulas by re-generating the latex from the parse tree, as well as tokenization.

Attribute	Explanation
image	Name of the image file containing the formula.
height	The height of the image in pixels.
width	The width of the image in pixels.
word2id_len	Length of the sequence after converting words in the formula to their respective IDs in a dictionary.
word2id	Sequence of dictionary ID numbers corresponding to the words or symbols in the LaTeX formula.
latex_ascii	The actual LaTeX formula in ASCII text, representing the mathematical formula depicted in the image.
padded_seq	The sequence of IDs padded to a fixed length for uniform processing.
seq_len	Original length of the sequence before any padding is applied.

Table 1. Attributes of *I2L-140K*.

Looking closely and analyzing the distribution of the dataset, we can see that the size of the images in the dataset varies greatly. The width of the images varies greatly from a minimum of only 3 pixels to a maximum of 1086 pixels

$$\int_{-\epsilon}^{\infty} dl\, \mathrm{e}^{-l\zeta} \int_{-\epsilon}^{\infty} dl' \mathrm{e}^{-l'\zeta} ll' \frac{l'-l}{l+l'} \{3\,\delta''(l) - \frac{3}{4} t\,\delta(l)\} = 0.$$

```
\[ \int_{-\epsilon}^{\infty} dl \,: \mathrm{e}^{-l\zeta}\int_{-\epsilon}
\mathrm{e}^{-l^{\prime}\zeta} ll^{\prime}\int_{-\epsilon}^{\infty} dl^{\prime}
\left\{ \frac{l^{\prime} - l}{l + l^{\prime}} \right\}\left
\{3 \, \delta^{\prime\prime}(l) - \frac{3}{4} t \, \delta(l)\right\} = 0 . \]
```

Figure 2. A training sample: the top is the input formula image and the bottom is the target sequence y

with an average width of about 521.49 pixels and a standard deviation of 221.75 pixels. On the other hand, the height of the image varies from 3 pixels to 126 pixels with an average height of about 64.55 pixels and a standard deviation of 22.29 pixels. This variability in image dimensions suggests a diverse set of images in terms of size.

Turning to the formulas, which are represented in LaTeX, the lengths (in characters) of these formulas also have a wide range. The shortest formula consists of only a single character, whereas the longest stretches to 595 characters. The average length of a formula is 162.1 characters, with a standard deviation of 80.63 characters. This indicates that while many formulas are of moderate length, there is a considerable range, including some very lengthy expressions. Furthermore, the analysis shows that 3.93% of the formulas are multi-lined. This suggests that while the majority of formulas are single-line formulas, a significant number of formulas employ more complex structures across multiple lines, indicating the diversity of the dataset we used.

Math Symbol	Count
$\int$	22044
∞	9141
$\sum$	15451
$\prod$	2102
$\sqrt{}$	17331
$n!$	6436
$\leftarrow$	6059
$\rightarrow$	66
$\frac{a}{b}$	135124
$\sum$	15451
$\equiv$	6706
$\binom{n}{r}$	229

Table 2. Examples of Math Symbols and Counts.

Looking more deeply into the content of the formulas, we can see that the symbols that appear are also diverse, totaling 518. The distribution of these symbols is not uniform, as some symbols are used more than others in mathematical formulas, and looking at the distribution of 12 of these symbols, we can see that it ranges from almost all formulas having the symbol to less than 1% of formulas having the

symbol. This could pose a potential problem for training, as the model may not have enough samples of the symbol to consistently interpret it correctly. An example of a training sample is shown in Figure 2.

2.2. Encoder

We explored a variety of encoder architectures to determine the most effective model for our needs. Thus, we specifically selected the newest and most promising architectures, including ResNet, ConvNeXt, and Swin Transformer.

The input to each encoder was a padded image with dimensions $(3, 1088, 128)$, where 3 represents the number of channels, 1088 is the width, and 128 is the height of the image. The output of the encoder is a stack of feature maps in $R^{C\times W\times H}$, where C represents the output feature channel and W and H represent the width and height of the feature map.

We used *torchvision* [10] to implement two encoders, the ConvNeXt and Swin transformers. This library simplifies the implementation of these models and includes pretrained versions of them that we have also tested. For ease of customization, our ResNet encoder is based on a previous implementation [11].

2.2.1 ResNet

Different sizes of ResNet [12] have been tested including: *resnet18*, *resnet34*, *resnet50*, *resnet101*, *resnet152*. Average pooling, flattening, and linear layer have been removed from the implementation because the output of ResNet will be used for the decoder and we only want to use ResNet as a feature extractor to extract features from the image.

2.2.2 ConvNeXt

Different sizes of ConvNeXt [2] have been tested including: *convnext-tiny*, *convnext-small*, *convnext-base* and *convnext-large*. Average pooling and the fully connected layer have been deleted from the model for the same reasons as in ResNet. In addition to these four standard implementations, we have implemented a customized ConvNeXt based on *convnext-tiny*. With this implementation, we can generate customized instances of ConvNeXt with different output vector dimensions.

2.2.3 Swin Transformer

For the same reason, for Swin Transformers [3], we omit the pooling, flatten and linear layers. The models tested were *swin-v2-t*, *swin-v2-t* and *swin-v2-b*.

2.3. Decoder

We also explored a variety of SOTA decoder architectures, including Transformer and GPT.

The decoder takes the flattened and transposed output of the encoder output $x_1 \in R^{HW\times C}$, and a sequence of tokens $x_2 \in N^L$ (where L is the length of the sequence) as input. It produces logits $y \in R^{(HW+L)\times V}$ as output (where V is the size of the vocabulary on which the decoder operates).

2.3.1 Transformer

The Transformer [13] architecture consists of an encoder that encodes the input (in our case the output of the encoder) and then uses an attention layer to perform cross-attention to the output embedding (in our case another input, a sequence of tokens). Finally, it predicts the next token for each token in the provided token sequence.

To implement the transformer model, we mainly rely on the transformer module provided in PyTorch [14]. On top of that, we implemented the embedding layer and positional encoding mentioned in [13] and added a generator module at the end to compute logits and losses.

Embedding Layer and Positional Encoding: The input $x_2 \in N^L$ is passed through the embedding layer to output embedding vectors $E \in R^{L\times D}$ (where D is the dimension of each embedding vector). Then, a pre-computed fixed positional encoding is added to E to add position information to the embedding vectors.

Transformer: The transformer module then uses $x_1 \in R^{HW\times C}$ as the input embedding and $E \in R^{L\times D}$ as the output embedding and performs a cross-attention between them, outputting $O \in R^{L\times D}$. The transformer module requires that $C = D$.

Generator: The generator takes $O \in R^{L\times D}$ as input, produces prediction Pred $\in R^{L\times V}$, and calculates the loss if the target is provided.

2.3.2 GPT

The GPT [5] is a decoder-only architecture that consists of a decoder that decodes the input and predicts the next token for each token in the input. Internally, it uses self-attention layer to perform attention on the input. In our implementation, we adapted the [15] implementation to prepend the encoder output to a list of embedding vectors, which are then fed as inputs to the transformer block.

Positional Encoding: Another difference between the GPT and the transformer is that the transformer uses a fixed positional encoding while the GPT uses a learnable positional encoding.

3. Challenges

3.1. Dataset Preprocessing

In the training workflow, we first load all the datasets as data frames and then pass them to the *CustomLatexDataset* class, which will fetch each image and its corresponding label as the training starts. Since the images and labels in the dataset vary in size, we had to preprocess them. We initially decided to process them on-the-fly, i.e., pad them when needed. However, it is not wise to do so because the same data will be used multiple times in different epochs and in training different models. To eliminate the need for repetitive preprocessing, we decided to preprocess all data before training and then simply load the preprocessed data on-the-fly.

The preprocessing of all the images (padding them with ones so that the images are the same size) is done in batches rather than all at once, as this is a very memory-intensive task. Due to disk size limitations, we cannot preprocess the labels ahead of time, so we must preprocess the labels on-the-fly in the *CustomLatexDataset* class: pad them with special padding tokens up to the size of 151, which is the size of the longest tokenized formula in the dataset. In the case of the transformer model, an additional start-of-sequence token is also added at the beginning of the label. We refer the reader to Appendix B for a detailed description of our tokenization strategy.

3.2. Training

Another challenge was the inability of our models to learn properly. After trying multiple combinations of encoders and decoders, the accuracy of all our models (with parameter counts ranging from a few million to a hundred million) was stuck at around 35% (GPT) and 15% (transformer).

Using PyTorch implementations with pre-trained weights was the first solution we tried. We tried this because we thought that the dataset we were operating on was not large enough for the model to learn how to extract meaningful information from the data. After a few attempts, we realized that the performance did not improve, but in fact got worse. One reason for this could be that the pre-trained weights were not suitable for our specific task, so this introduced overhead as our model needed to forget and re-learn new weights.

The second solution we tried was to reduce the dimension of the embedding vectors. This was based on the intuition that the model might be too complex for our task, resulting in the inability to extract meaningful representations. Especially in our case, since our dataset is not very large, a less complex model with fewer degrees of freedom may be better for making predictions in regions with little or no data. Therefore, we decided to reduce the dimension of the embedding vectors. As a result, the accuracy of our model with GPT decoder was significantly improved (by almost a factor of 2).

The third solution we tried specifically for the transformer was to apply proper initialization. We realized that we had not properly initialized the embedding and fully connected layers previously. After making these changes, the accuracy of the transformer architecture improved from 15% to about 30%.

4. Experiments and Results

4.1. Evaluation Metrics

There are 4 metrics used to measure the effectiveness of our model.

Accuracy: Standard multiclass accuracy using micro-averaging and ignoring the index of the padding token.

BLEU [16]: Compare the n-gram (1,2,3,4) of the machine-generated text with the n-gram of the reference text to derive a score, which is then multiplied by a brevity penalty to account for translations that are too short to arrive at a final BLEU score.

Perplexity [17]: Exponent of the cross-entropy loss of the model.

Number of weights: This is an implicit measure of success because the smaller the weights, the faster the model runs locally.

4.2. Loss function

Cross-entropy [18] is used as the loss function of our model. It is suitable for classification tasks like ours where the output for each predicted token is a probability distribution across the vocabulary. The loss increases as the predicted probability diverges from the actual label.

Given a true distribution p and an estimated distribution q, the cross-entropy between the distributions is defined as:

$$H(p, q) = -\sum_{x} p(x) \log q(x)$$

In our context, p represents the true distribution, where the true character class has a probability of 1 and all others 0 and q is the predicted probability distribution over the vocabulary by our model. When training our model, we calculate the cross-entropy loss for each token, and then sum and average it to get the final loss.

4.3. Optimizer

The AdamW [19] optimizer was used for training. It is an improvement on the Adam optimizer that improves training and generalization by decoupling weight decay from gradient updating. Optimizers such as Adam use weight decay as part of the gradient update, which can lead to

less efficient regularization, especially in the case of adaptive learning rates. AdamW improves on this by applying weight decay directly to the weights in addition to the gradient update.

4.4. Experiment Results

Multiple combinations of encoder and decoders were tested. We struggle to get good results until reducing the embedding dimension. The first 4 rows of the Table 3 show the result before the dimensionality reduction, and the last row shows the result of our final best model, which achieves 65% accuracy with the customized ConvNeXt encoder (which generates embedding of dimension 64) and the GPT decoder.

By comparing different rows, we can see a significant increase of accuracy and a decrease on the perplexity in the last row of Table 3. This is our main finding in this paper, namely that in some use cases (i.e., when the vocabulary is not large), the decoder does not need as large an embedding. The reasons why this can happen are discussed in 3.2.

We also evaluated the best performing model for its autoregressive ability and measured its BLEU score on the test dataset. We used the beam search [20] with a beam size 2 to reduce the probability of the model being trapped at suboptimal solutions when making predictions. However, during the evaluation process, we found that the model always outputs the same formula no matter what the input image is. We hypothesize that this occurs because the CNN is not powerful enough to extract useful features from the image (things in the center of the image).

To test our hypothesis, we loaded two random examples from the dataset and found that the cosine similarity between these two outputs of the encoder is 1, which means that the encoder produces exactly the same output for these two different images, which explains why the decoder decodes the same formula every time. [21] gives a possible explanation: padding the image with 1 (white) will adversely interfere with the optimization of the kernel's weights, thus preventing the CNN from learning how to correctly extract the information in the image. The recommended solution, suggested by [21], is to pad the image with 0 as it is proven that this does not adversely affect backward propagation and speeds up forward propagation.

5. Discussion

5.1. Ablation Study

To further investigate how encoder and decoder contributes to the performance of our model, we performed an ablation study. The results are shown in Table 4.

As can be seen from the table, each transformer block in the decoder contributes significantly to the accuracy of the prediction, since removing a block from the decoder results in a 7-8% decrease in accuracy.

However, for ConvNeXt, we can see that removing 1 or even 2 ConvNeXt Blocks does not change the accuracy of the overall model, which means that the ConvNeXt encoder has not yet learned how to properly extract features from images. This finding further confirms our hypothesis in 4.4.

5.2. Limitations and Future Work

Currently, *LaTeXify* is unable to operate autoregressively in a meaningful way because of the poor performance of the encoder and therefore cannot be used for any practical applications. Future work on the *LaTeXify* project will focus on addressing poor encoder performance, starting with trying to pad the image with zeros instead of ones, while keeping the model small enough to run locally. Our goal is to release the model so that everyone can convert images to formulas in a reliable way for free.

6. Work Division

The contributions from each team member are provided in Table 5.

Model	Train Acc.	Perplexity	Number of Weights (in millions)	Embedding Dimension
resnet50-gpt	0.34	22.19	328.10	2048
convnext_tiny-gpt	0.35	21.35	92.9	768
swin_v2_b-gpt	0.34	21.84	71.03	1024
convnext_custom-transformer	0.29	256.1	0.97	64
convnext_custom-gpt	**0.65**	**4.26**	**0.90**	64

Table 3. The performance of *LaTeXify* model. The best performance is indicated in bold.

Ablation	Test Acc.
	0.6294
-1 Transformer Block	0.5481
-2 Transformer Blocks	0.4789
-1 ConvNeXt Block	0.6294
-2 ConvNeXt Blocks	0.6294

Table 4. The result of the ablation study. The ablation is based on the best performing model convnext_custom-gpt.

References

[1] [Online]. Available: https://mathpix.com/ 1

[2] Z. Liu, H. Mao, C. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, "A convnet for the 2020s," in *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2022, pp. 11 966–11 976. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CVPR52688.2022.01167 1, 3

[3] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," in *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*. Los Alamitos, CA, USA: IEEE Computer Society, oct 2021, pp. 9992–10 002. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICCV48922.2021.00986 1, 3

[4] N. Pang, C. Yang, X. Zhu, J. Li, and X.-C. Yin, "Global context-based network with transformer for image2latex," in *2020 25th International Conference on Pattern Recognition (ICPR)*, 2021, pp. 4650–4656. 1

[5] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019. 1, 3

[6] S. S. Singh, "Teaching machines to code: Neural markup generation with visual attention," 2018. 1, 2

[7] L. Blecher, "pix2tex: Using a vit to convert images of equations into latex code," https://github.com/lukas-blecher/LaTeX-OCR#References, 2023. 1

[8] Y. Deng, A. Kanervisto, J. Ling, and A. M. Rush, "Image-to-markup generation with coarse-to-fine attention," in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML'17. JMLR.org, 2017, pp. 980–989. 1

[9] A. Kanervisto, "im2latex-100k , arxiv:1609.04938," Jul. 2016. [Online]. Available: https://doi.org/10.5281/zenodo.56198 2

[10] T. maintainers and contributors, "TorchVision: PyTorch's Computer Vision library," Nov. 2016. [Online]. Available: https://github.com/pytorch/vision 3

[11] Karunesh, "How to resnet in pytorch," https://medium.com/@karuneshu21/how-to-resnet-in-pytorch-9acb01f36cf5, 2023, accessed: 2023-04-29. 3

[12] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015. 3

[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023. 3

[14] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, G. Chauhan, A. Chourdia, W. Constable, A. Desmaison, Z. DeVito, E. Ellison, W. Feng, J. Gong, M. Gschwind, B. Hirsh, S. Huang, K. Kalambarkar, L. Kirsch, M. Lazos, M. Lezcano, Y. Liang, J. Liang, Y. Lu, C. Luk, B. Maher, Y. Pan, C. Puhrsch, M. Reso, M. Saroufim, M. Y. Siraichi, H. Suk, M. Suo, P. Tillet, E. Wang, X. Wang, W. Wen, S. Zhang, X. Zhao, K. Zhou, R. Zou, A. Mathews, G. Chanan, P. Wu, and S. Chintala, "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation," in *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, Apr. 2024. [Online]. Available: https://pytorch.org/assets/pytorch2-2.pdf 3, 8

Student	Contribution	Details
Alejandro Canada	Environment configuration, data pre-processing, custom dataset class, encoder implementation, experimentation.	Set up all the requirement environment needed integrating GitHub in Colab to facilitate development. After multiple tries finding the best strategy to fit the data on memory. Preprocess images on a separate notebook and labels. Modify implementation of data handler and data loader classes. Implement two encoder, ResNet and ConvNeXt (then changed for PyTorch implementation). Implementation of training loop with Yuxiang. Train multiple models trying to find the best hyperparameters.
Pablo San Francisco	Data loading, handling and visualization, tokenizer class implementations, and experimentation.	Implemented the data loader class, which loaded the dataset in a Colab enviroment, as well as the data handler, which further processed the data from the dataset. Did visualizations and got statistics to further understand the data before the model training. Implemented the tokenizer class for the model and experimented with several encoders and decoders, training them to optimize the final results for the model.
Yuxiang Qiu	Implementation of the encoder, decoder, evaluation metrics, generating functions, and experimentation.	Implement/Modify encoders, including ResNet, ConvNeXt, and Swin Transformer. Implement decoders based on the PyTorch's transformer architecture and the OpenAI's GPT-based decoder-only architecture. Implement the beam search to test the generating capability of our model. Implement the metrics (*BLEUScore*, *Perplexity*, and *MulticlassAccuracy*) to evaluate the performance of our model. Experiment with transformer and decoder models to find the optimal architecture and hyperparameters and find that the embedding dimension affects the performance of our models. Conduct performance analysis and ablation study to understand how each part of the *LateXify* model contributes to the overall performance.
All	Meeting, Paper Review, Report Writing.	Includes: Multiple meetings per week. Review 3 papers. Report writing.

Table 5. Contributions of team members.

[15] minGPT maintainers and contributors, "mingpt," https://github.com/karpathy/minGPT, 2020. 3, 8

[16] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ser. ACL '02. USA: Association for Computational Linguistics, 2002, p. 311–318. [Online]. Available: https://doi.org/10.3115/1073083.1073135 4

[17] F. Jelinek, R. L. Mercer, L. R. Bahl, and J. K. Baker, "Perplexity—a measure of the difficulty of speech recognition tasks," *The Journal of the Acoustical Society of America*, vol. 62, no. S1, pp. S63–S63, 08 2005. [Online]. Available: https://doi.org/10.1121/1.2016299 4

[18] C. E. Shannon, "A mathematical theory of communication," *The Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948. 4

[19] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," 2019. 4

[20] M. Freitag and Y. Al-Onaizan, "Beam search strategies for neural machine translation," in *Proceedings of the First Workshop on Neural Machine Translation*. Association for Computational Linguistics, 2017. [Online]. Available: http://dx.doi.org/10.18653/v1/W17-3207 5

[21] M. Hashemi, "Enlarging smaller images before inputting into convolutional neural network: zero-padding vs. interpolation," *Journal of Big Data*, vol. 6, no. 1, pp. 1–13, 2019. 5

[22] Nicki Skafte Detlefsen, Jiri Borovec, Justus Schock, Ananya Harsh, Teddy Koker, Luca Di Liello, Daniel Stancl, Changsheng Quan, Maxim Grechkin, and William Falcon, "TorchMetrics - Measuring Repro-

ducibility in PyTorch," Feb. 2022. [Online]. Available: https://github.com/Lightning-AI/torchmetrics 8

[23] T. maintainers and contributors, "Torcheval," https://github.com/pytorch/torcheval, 2022. 8

A. Project Code Repository

Our code base is divided into two main directories: *src* and *test*. The first directory contains the source code that implements our models (encoders and decoders), metrics, training and generation functions. The second directory contains the unit test cases we have written for our models and metrics. I will discuss the *src* folder in more detail below:

- **encoder**: This subfolder contains our implementations of ResNet, ConvNeXt and Swin Transformer.
- **decoder**: This subfolder contains our implementation of the transformer (based on PyTorch [14]) and GPT-based decoder-only architecture (based on *minGPT* [15]).
- **analysis**: This sub-folder contains the metrics used for our analysis. In it, we re-export the *MulticlassAccuracy* class [22], and the *Perplexity* class [23]. Based on the *BLEUScore* class in [23], we fixed a bug in the brevity penalty calculation and designed our class for calculating the BLEU score.
- **train.py**: This file contains the code to train the model.
- **models.py**: This file contains the definition of our *LatexifyModel*.
- **generate.py**: This file contains an implementation of the beam search and *generate* functions that generate text from the input formula image.
- **data_handler.py** and **data_loader.py**: These two files implement the data loading functions and the interface *Dataset* defined by PyTorch, which makes it easy to integrate the dataset into the training process.

The Github repository for our final project is at: `https://github.com/Alex44lel/latexify1.0`.

B. Tokenization

We used two different tokenization strategies based on the decoders used by our model.

For the transformer decoder, we added three additional tokens to the vocabulary: a start-of-sequence token, a end-of-sequence token and a padding token. The reason we need the start-of-sequence token is that we need to use a token at the beginning to perform cross attention with the embedding vectors extracted from the image, which we can then use to instruct the model to autoregressively generate the formula text. As the name implies, the end-of-sequence token signals to the model that the input ends. We separate it from the padding token because we ignore the padding tokens when calculating the loss, and if we just use a padding

token to mark the end of the generated text, the model may not know how to express the end of the equation, which can cause trouble when it is used autoregressively later on.

For the GPT decoder, we added only two tokens to the vocabulary: an end-of-sequence token and a padding token. We did not add a start-of-sequence token because the model output at the position of the last prepended hidden input vector can be considered as a prediction of the first token that the model should output. Therefore, we don't need an additional start-of-sequence token to instruct the model to output the formula text.