LaTeXify - OCR for MATH

Prabhat Reddy, Ninad Gandhi, Munish Monga
C-MInDS, Indian Institute of Technology, Bombay, India
Roll Numbers: 22M2156, 22M2151, 22M2153

***Abstract*—Optical Character Recognition (OCR) is a technology that converts printed/handwritten text from images to machine encoded text, and finds important applications across many applications such as document digitization, word processing and banking. Application of OCR for mathematical expressions is a harder problem as compared to natural language text, as mathematical expressions aren't written in a linear fashion. While OCR for printed text is considered a solved problem, OCR for handwritten text can still be improved upon. The goal of the project is to explore the current status of Mathematical OCR for handwritten text, and develop a machine learning model that takes images of handwritten mathematical expressions (with reasonably legible handwriting) as inputs and generates equivalent LaTeX representations for the same. We have identified a research paper, implemented the proposed model and created an initial prototype to generate LaTeX expressions from computer-rendered math expressions. The model follows an encoder-decoder architecture, where the decoder is a LaTeX langauge model. The encoder consists of two components, which encode the rows of the input image and the mathematical symbols estracted from the image simultaneously and concatenates them together. Future work can include improving the model and evaluating it on a variety of datasets, before integrating it into an OCR application that can convert an image of an entire document, containing both text and math expressions, into a LaTeX document.**

***Index Terms*—Mathematical OCR, Deep Learning, Graph Neural Networks, Encoder-Decoder model, Image to LaTeX**

I. Introduction

Mathematical expressions are an integral part of research papers across various research domains and help formulate a real-world concept into mathematics. Writing out mathematical expressions by hand is easy and intuitive, but typing them down on a computer is a very time taking task. While it's difficult for beginners to type out LaTeX mathematical expressions owing to the large learning curve of the work, those who are proficient with LaTeX take plenty of time to type out papers in LaTeX as well. We believe that developing a technology that can transform handwritten research papers to mathematical documents can save a significant amount of researchers' time and allow them to focus more on their research.

Our project is focused on arguably the most difficult task in this problem - converting handwritten mathematical expressions into LaTeX text. We develop a deep-learning model with an encoder-decoder architecture that can be trained end-to-end, which takes handwritten mathematical text as images and give out its equivalent LaTeX representation.

In Section II we go through the previous work done in the domain, primarily aimed at the task of rendering printed mathematical expressions. Section III formalizes and discusses the problem of creating the desired model, Section IV discusses the implemented model at a greater depth. Section V discusses the dataset and model hyperparameters used for training and section VI discusses the performance of the model. Finally, section VII concludes the report and hints at possible work that can be done to improve the model.

II. Related Work

The problem of converting text to digital markup is quite old, even before machine learning systems were the norm. There has been extensive work done in this area, digitization of printed mathematical expressions in particular, and many research papers have been written to tackle the image to LaTeX problem. Initial systems to tackle this problem were rule based, the most popular one being [5]. While developing rule-based systems could be a possible solution for printed text, analysis of handwritten text is a much more complex task owing to its inconsistent structure.

A deep learning based model to recognize typeset mathematical from manually extracted symbols or *connected components* is proposed by [3], although the model was aimed at generating any general markup. An end-to-end trainable deep learning model using and encoder-decoder architecture and course-to-fine attention to convert mathematical expressions to LaTeX was built in [1]. The encoder consists of a Convolutional Neural Network (CNN) for feature extraction, followed by a Long Short Term Memory (LSTM) network that encodes each row of the extracted features. These encoded features are further fed to a decoder LSTM, which is a LaTeX language model, through a coarse-to-fine attention mechanism.

Inspired from the work mentioned above, a network that uses both extracted symbols and row encoded features is proposed in [4]. The novel idea of this paper is to encode the extracted symbols and feed them through a Graph Neural Network (GNN) so that spatial information of the formula elements can be taken into account by the model. We have implemented the model proposed by [4], making suitable design choices along the way.

III. Problem Formulation and Evaluation Metrics

A. Problem Formulation

Converting a sequence of LaTeX tokens to its corresponding image is done through a LaTeX rendering engine. Note that many different LaTeX sequences could lead to the same rendered output because of how one can generate LaTeX tokens. Note that the following mathematical expression,

$$y = x_i^2 + w$$

has at least two valid LaTeX input strings; ”y=x^2_i+w” and ”y=x_i^2+w”. That is, we can swap subscripts and superscripts. Moreover, we can have even more possible sequences for this expression if we bring curly braces and white spaces into the picture. So, many LaTeX strings can generate the same output, and hence the LaTeX engine is modeled by a `compile` function, is a **many-to-one function** given by

$$\texttt{compile}(y) = x, \tag{1}$$

where $y = \{y_1, y_2, ..., y_T\}$ is a series of LaTeX tokens and x is its corresponding rendered image. Note that a many-to-one function does not have a unique inverse, so we can obtain two distinct series of tokens y_1 and y_2 such that $\texttt{compile}(y_1) = \texttt{compile}(y_2)$. The goal is to obtain an inverse of the `compile` function, which we call `model` such that

$$\texttt{model}(x) = \hat{y}, \tag{2}$$

where $\hat{y}$ is a possible token for x i.e. $\texttt{compile}(\hat{y}) = x$. We want to use machine-learning techniques to create such a model. The fact that the LaTeX engine is not a bijection but a many-to-one function means that the inverse function is not unique. That is, the machine-learning model could learn any one of the inverse functions from the whole space of possible inverse functions of the LaTeX engine.

B. Evaluation Metrics

So the question is how do you evaluate your model? The problem here is that the LaTeX representation of an input image is not unique and there are multiple right answers. In such a case, how does one decide whether the prediction generated by the model is indeed correct? A few metrics that answer this question in various degrees are discussed below:

- *Exact Match (Re-render and compare)*: The idea here is to bypass the problem of the rendering function being many-to-one. See that the problem stems from the fact that many LaTeX strings generate the same rendered image. So, we can just render the true label and the predicted string and compare the renders pixel by pixel. This way, the subscript-superscript sequence, presence or absence of curly braces and white spaces becomes irrelevant. Evaluation is done between $\hat{x} = \texttt{compile}(\hat{y})$ and x such that $\hat{x} \sim x$ i.e. the aim is to produce similar rendered images while $\hat{y}$ may or may not be similar to the ground-truth markup y. This approach is not feasible though. The dataset contains 100,000 images and it becomes computationally inefficient to perform renderings for each and every input sample and corresponding prediction. Hence, this approach is not used to score the model.

- *Exact Match of tokens*: This compares the generated sequence of tokens to its corresponding target. Let $Y = \{y_1, y_2, ...y_N\}$ and $\hat{Y} = \{\hat{y}_1, \hat{y}_2, ...\hat{y}_N\}$ be the set of N LaTeX expressions and the set of corresponding rendered images respectively. Then

$$EM(Y, \hat{Y}) = \frac{1}{N}\sum_{i=1}^{N} I_{y_i,\hat{y}_i}, \tag{3}$$

where $I_{y_i,\hat{y}_i} = 1$ if $y_i, \hat{y}_i$, and 0 otherwise.

- *Edit Distance*: The edit distance between the ground truth and prediction is defined as the number of characters/tokens required to be edited to convert the prediction to the ground truth.

- *BLEU-4*: BLEU-4 scoring is widely used to evaluate language models. The basic idea is to look at 1-grams, 2-grams, 3-grams, and 4-grams of ground truth and prediction and compare them. The overall score is the weighted average of the four scores obtained in the previous step.

IV. Model and Implementation Details

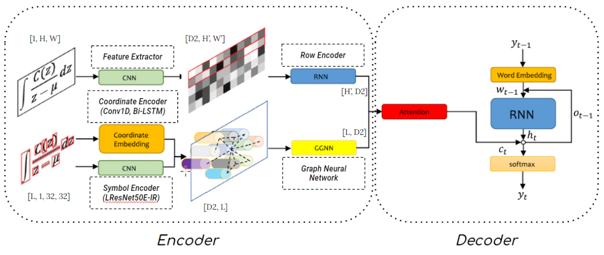

Fig. 1. Model Pipeline [4]

We have implemented an encoder-decoder architecture as a solution. The encoder part has three main components. First, a Row encoder which looks at the overall image and encodes information about the overall formula. Second, there is a Coordinate embedding unit that takes in the coordinates of each symbol in the formula image and maps it to a higher-dimensional feature space. Third, we have a symbol encoder which is a CNN. The symbol encoder takes in images of each symbol and encodes it into a vector of size $L \times D_1$, where L is the number of symbols in the image and D_1 is a hyperparameter. The Row encoder looks at the image on a larger scale and encodes information at the formula level. Alternately the symbol encoder and the coordinate embedding encode the information at a smaller scale than individual symbols. The outputs of the coordinate embedding layer and symbol encoder are concatenated to feed as input to a Graph Neural Network. The GNN also takes a graph of symbols. The symbol graph is generated using a line-of-sight algorithm, which establishes an edge between two symbols they are in line-of-sight of one another. The output of Row encoder and the GNN are further concatenated to create the final encoding of the formula image. Now, the decoder comes into the picture. The decoder start with an attention mechanism unit, followed by a standard RNN decoder model which is a Language model for LaTeX.

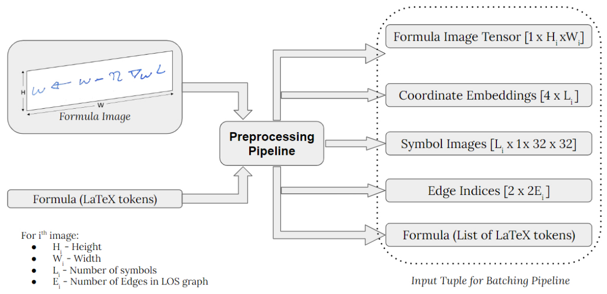
H
W
Formula Image
Preprocessing Pipeline
Formula (LaTeX tokens)
Formula Image Tensor [1 x H_i xW_i]
Coordinate Embeddings [4 x L_i]
Symbol Images [L_i x 1x 32 x 32]
Edge Indices [2 x 2E_i]
Formula (List of LaTeX tokens)
Input Tuple for Batching Pipeline
For i^th image:
H_i - Height
W_i - Width
L_i - Number of symbols
E_i - Number of Edges in LOS graph

Fig. 2. Pre Processing Pipeline

A. *Preprocessing Pipeline*

We must preprocess the raw input data before we feed it into the model. Our preprocessing pipeline generates a 5-tuple for every input image, which will be batched and given to the model later on. This 5-tuple consists of the following items:

- *Formula Image*: This is the raw image itself. Its dimensions are $[1 \times H_i \times W_i]$, where H_i and W_i are the height and width of the input raw image respectively.
- *Coordinates*: It contains the spatial information of each symbol extracted from the raw image. It is a 2D array of shape $[4 \times L_i]$, where L_i is the number of symbols extracted from the image. There are 4 parameters for each symbol, namely spatial coordinates of centroid (x, y), the height h, and the width w of the image.
- *Symbol Images*: This is an array of scaled images for each symbol. Each symbol image has size $[1 \times 32 \times 32]$, and the total symbol images array has size $[L_i \times 1 \times 32 \times 32]$.
- *Edge Indices*: This contains the number of edges of the undirected LOS graph, which is extracted from the raw image using the coordinates of symbols. Its dimensions are $[2, 2E_i]$, where E_i is the number of edges in the undirected graph. We have $2E_i$ here because we represent every undirected edge as a combination of 2 directed edges (i.e. if the graph has edge (u, v) then it must also have edge (v, u).
- *Formula*: This is a list of tokens that are passed as labels, along with the raw images, to the preprocessing pipeline.

The preprocessing pipeline consists of the following algorithms to extract coordinates, symbol images and edge indices from the formula image.

1) Symbol Segmentation: To generate *symbol images* and *coordinates*, the first step is to identify all the symbols in the image. See that there are algorithms already designed to identify symbol images. Specifically, we use the OpenCV library to perform this step. First, we convert the RGB image to grayscale, and then threshold it to convert it to binary form, where an image pixel is either white or black, i.e. it is either one or zero. We use Otsu's thresholding algorithm which uses the pixel intensity histogram to threshold the image. Now, on this thresholded image, we can perform the connected component analysis. The connected component algorithm creates a bounding box around all the symbols in the image. It looks at the set of pixels that are adjacent to each other and identifies it as a symbol. A sample input is used as an example in the below images to illustrate how the connected component algorithm works.

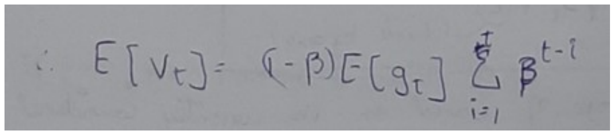

Fig. 3. Input Raw Image

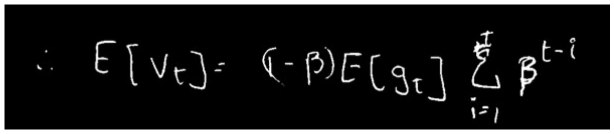

Fig. 4. Thresholded Image

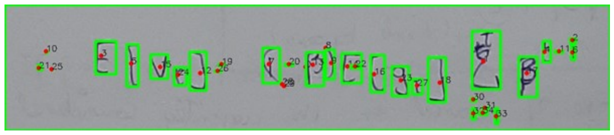

Fig. 5. Bounding Boxes and Centroids

2) Line-Of-Sight Algorithm: The next step is to create a graph out of the segmented symbols so that the set of edges can be fed into the GNN. This idea is inspired from [3]. The Line-Of-Sight Graph creation algorithm is as follows:

Algorithm 1 LOS Algorithm

```
1: for each symbol-node do
2:     for each direction do
3:         Look at the pixels in a line in the current direction starting the from the centroids of current symbol until you run out of the image or you hit a white pixel (because the thresholded image is white on black)
4:         if run out of image dimension then
5:             Look in the next direction
6:         end if
7:         if hit a white pixel then
8:             if hit a pixel of the original symbol then
9:                 Continue searching
10:            else
11:                You have hit a new symbol. establish an edge between these two symbols and move to the next direction
12:            end if
13:        end if
14:    end for
15: end for
```

There are many problems with the LOS graph algorithm. First, it is computationally very inefficient. Second, the choice of the number of directions determines how accurately the resultant graph can be built. Sometimes, we might want some symbols to connect to each other in the LOS graph, but the restriction on the number of directions limits how many edges can be discovered. But then again, increasing the number of directions increases the computational complexity heavily. Note in the following examples that the three dots in the "therefore" symbols are not connected because the number of exploration directions is not enough to establish an edge.

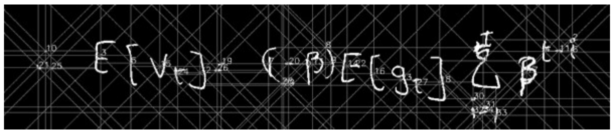

Fig. 6. LOS Exploration for each node

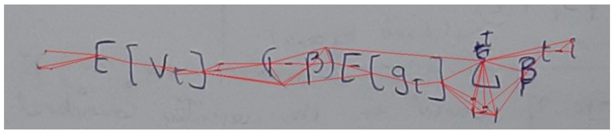

Fig. 7. LOS Graph for input image

A very simple alternative to the line-of-sight algorithm is given in Algorithm 2.

Algorithm 2 LOS Graph (alternative)

```
1: for each symbol-node do
2:     Look at a square region around the symbol and establish an edge between all the symbols having their respective centroids within this region
3: end for
```

See that this algorithm is computationally more efficient than the LOS algorithm and even generates much more densely connected graphs in practice.

B. Batching Pipeline

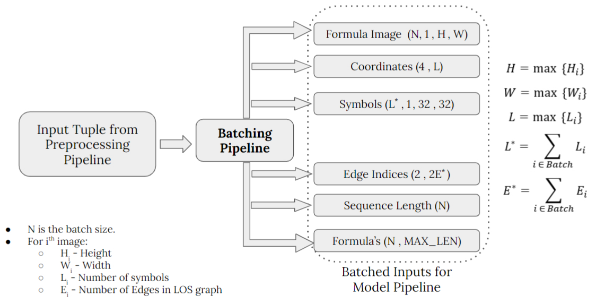
Input Tuple from Preprocessing Pipeline
Batching Pipeline
Formula Image (N, 1, H, W)
Coordinates (4, L)
Symbols (L^*, 1, 32, 32)
Edge Indices (2, $2E^*$)
Sequence Length (N)
Formula's (N, MAX_LEN)
Batched Inputs for Model Pipeline
$H = \max\{H_i\}$
$W = \max\{W_i\}$
$L = \max\{L_i\}$
$L^* = \sum_{i \in Batch} L_i$
$E^* = \sum_{i \in Batch} E_i$
N is the batch size.
For ith image:
H_i - Height
W_i - Width
L_i - Number of symbols
E_i - Number of Edges in LOS graph

Fig. 8. Batching Pipeline

Batching helps us process multiple inputs simultaneously decreases overall processing time. Increasing the batch size also helps reduce noise in the weight update at every time step in gradient descent. Note that batching is done differently for each kind of input:

- Formula Image and Formulas: The inputs are stacked along a new dimension, which is a general practice when training CNNs.
- : Coordinates: The inputs are first padded with a constant before stacking them along a new dimension. This is done because each image has a variable number of symbols, and stacking of tensors of different length isn't possible.
- Symbol Images: The inputs here are concatenated along the first dimension, as opposed to stacking along a new dimension. This is because all the symbols can be processed parallelly and there is no interaction between symbols.
- Edge Indices: All the edge indices are concatenated along the second dimension, which essentially groups multiple graphs into a single graph containing *unconnected* components. PyTorch Geometric, which is the library used to implement the GNN, recommends batching inputs like this.

We also generate another tensor called *sequence length* in order to assist batching within the model.

C. Model Pipeline

As we have seen, the proposed model has an encoder-decoder architecture. In the preprocessing pipeline, we derive various elements from the input image like coordinates of symbols, symbol image of size $1 \times 32 \times 32$, edges of a Line-of-sight graph from the spatial information of symbols, formula region row encoding. Let us now look at how each of these inputs is fed into different parts of the encoder and processed to produce coordinate embeddings, symbol encodings, and formula encodings.

1) Row Encoder: Row encoder is a combination of a convolutional neural network followed by a Recurrent neural network. Input to the Row encoder is the formula image of size 1xHxW which is one of the elements of the 5-tuple generated by the preprocessing pipeline. The CNN extracts visual features from the overall image and outputs a vector of size $D_2 \times H' \times W'$. This vector becomes the input to the RNN which looks at the sequence of rows of dimension $D_2 \times 1$. There are $H' \times W'$ such rows and correspondingly $H' \times W'$ hidden states are generated by the RNN. We only use the last hidden state of the RNN as the output which is of the size D_2'. This becomes one of the two inputs to the attention mechanism.

2) Coordinate Embedding: Recall that the preprocessing pipeline outputs a vector of size $4 \times L$ which contains coordinates of each of the L symbols: centroids, width and height which is a 4-tuple. The coordinate encoder maps the coordinate input into a higher-dimensional feature space with 1D convolutions on each feature. This converts the 4XL features into a vector with dimension $\frac{D_1}{2} \times L$. This is followed by some activation layers and a bidirectional LSTM. The Bidirectional LSTM takes the $\frac{D_1}{2} \times L$ features and converts them to $D_1 \times L$ encoding. This is the final output of this module.

3) Symbol Encoder: The CNN architecture for the symbol region is based on LResNet50E-IR. Symbol encoder gets L images of size $1 \times 32 \times 32$. That is, an input vector of size $L \times 1 \times 32 \times 32$. The symbol encoder is a CNN that generates symbol embeddings of dimension $D_1 \times L$.

4) GNN: We use a Gated Graph Neural Network [2] in this model, as suggested by [4]. The graph neural network follows the symbol encoder CNN and coordinate embedding module. The input to the GNN is made up of the embedding vectors generated by both these stages and the LOS graph generated using the LOS algorithm. See that the Coordinate encoder generates embeddings of size $D_1 \times L$ and the Symbol encoder generates embeddings of size $D_1 \times L$. These two vectors are concatenated to get a vector of size $D_2 \times L$ where each of the vector of size D_1 forms a node embedding of the corresponding symbol. The GNN runs two iterations which means each node in the graph influences a node that is at a distance of 2 from it. GNN takes symbol encodings of size $D_2 \times L$ and generate the output of size $D_2 \times L$. The computation of GNN is governed by the following equations:

$$m_{ij} = f^*(h_i^t, h_j^t, e_{ij}) \tag{4}$$

$$m_i = \sum_j m_{ji} \tag{5}$$

$$h_i^{t+1} = GRU(h_i^t, m_i) \tag{6}$$

The output of GNN is as follows:

$$h_G = \tanh(W^G[h^T, h^0]) \tag{7}$$

The output of the GNN and Row encoder are then concatenated to feed as input to the Attention layer discussed next.

5) Decoder with Soft Attention: The decoder is a LaTeX language model that is governed by the following equations:

$$h_t = LSTM(h_{t-1}, [w_{t-1}, o_{t-1}]) \tag{8}$$

$$o_t = \tanh(W^C[h_t, c_t]) \tag{9}$$

where h_t is the hidden state of LSTM unit, w_t is the token embedding and c_t is the context vector generated by the soft attention mechanism, which is as follows:

$$a_{i,t} = \beta^T \tanh(W_1 h_t + W_2 v_i) \tag{10}$$

$$\alpha_{i,t} = softmax(a_{i,t}) \tag{11}$$

$$c_t = \sum_i^{L+H'} \alpha_{i,t} v_i \tag{12}$$

TABLE I
IM2LaTeX Dataset

Split	Number of samples
Training	75,275
Validation	8,370
Test	10,355
Total	94,000

V. Dataset and Hyperparameters

We use the IM2LaTeX100K dataset, that was collated by [1], to train and evaluate the model. The dataset consists of LaTeX expressions extracted from many mathematical research papers, and it's corresponding rendered images. We use a preprocessed form of the dataset, which is available on Kaggle. The number of samples in the dataset along with the training, validation and test splits are given in Table I.

We trained our model for 30 epochs on the training dataset with a batch size of 32. We use a learning rate scheduler with an initial learning rate of 0.0003, which decays by a factor of 0.5 every time the validation loss stagnates. We set the embedding dimension hyperparameters $D_1 = 256$ and $D_2 = 512$ as suggested in [4]. We use the *normalized crossentropy loss* as our loss function.

VI. Evaluation

After training our model with the previously specified hyperparameters, we evaluated our model on the test split of the dataset. The model achieved a BLEU score of **35.71** and an edit distance of **42.26**, which is quite less as compared to the metrics projected in the paper. We believe this is the case because we haven't implemented a beam search strategy for evaluation as they have done in the paper.

TABLE II
Comparing our model performance with other models

Model	BLEU score	Edit distance
Deng et al. [1]	58.12	68.99
Peng et al. [4]	65.09	49.31
Ours	35.71	42.26

Here are some predicted tokens along with their corresponding reference tokens:

1) Reference:
```
f _ { i j } = \partial _
{ i } a _ { j } ^ { \mathrm
{ s } } - \partial _ { j }
a _ { i } ^ { \mathrm { s } } ,
```
Predicted:
```
f _ { i } \partial _ { } a _
{ } \partial _ { i } \partial
_ { - } \partial _ { j } a _
{ a } - \partial _ { j } \partial
_ { i } ^ { a } _ { j } ^ { a }
- i _ { j } ^ {
```
2) Reference:

```
2 \kappa _ { 1 1 } { } ^ { 2 }
T _ { 3 } { \tilde { T } } _
{ 6 } = 2 \pi n
```

Predicted:

```
2 \kappa _ { 1 } T _ { 2 } T _
{ 2 } T _ { 1 } T _ { 2 } T _ {
2 } T ^ { 2 } \tilde { T } _ {
2 } T _ { 2 } T _ { 2 } T _ { 2
} T _ { 2 }
```

3) Reference:

```
U ( \vec { x } ) = { \mathrm e
} ^ { { \mathrm i } \hat { r }
\cdot \vec { \tau } f ( r ) }
```

Predicted:

```
U ( \vec { \mathrm { r e } } )
= \mathrm { e } ^ { \mathrm {
U V } } \mathrm { e } ^ { \mathrm
{ i } \vec { \tau } \mathrm { e }
} f ( r ) = 0
```

We observe that the initial part of the prediction matches well with the reference token, but fails to generate good results after it breaks once. We believe that using beam search could help in alleviating errors cause by generating a wrong token at one time step.

VII. Conclusion and Future Work

We have developed a model for handwritten text recognition, but we weren't able to test it on a wide range of handwritten samples. The next order of work is to fine tune the model on collected samples of handwritten text and see how well it performs on unseen handwritten samples.

Acknowledgment: We thank Prof. Preethi Jyothi for teaching the course and giving us an opportunity to spend time on an involved project and learn lots of things along the way. We also thank the TA's for successfully managing such a large class while keeping the work standard to a maximum. We also thank the C-MInDS office that provided us a space to work and have lengthy discussions about the subject that broadened our understanding.

References

[1] 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.

[2] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[3] Mahshad Mahdavi, Michael Condon, Kenny Davila, and Richard Zanibbi. Lpga: Line-of-sight parsing with graph-based attention for math formula recognition. In *2019 International Conference on Document Analysis and Recognition (ICDAR)*, pages 647–654. IEEE, 2019.

[4] Shuai Peng, Liangcai Gao, Ke Yuan, and Zhi Tang. Image to latex with graph neural network for mathematical formula recognition. In *International Conference on Document Analysis and Recognition*, pages 648–663. Springer, 2021.

[5] 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.