ECEN 5060 (Deep Learning) Final Project Report

Haya Monawwar and Sungjoo Chung

May 6

1 Introduction

1.1 Overview

Handwritten mathematical expressions (HMEs) are indispensable in various domains, such as engineering, education, and science. The development of pen-based or touch-based devices has provided a user-friendly interface to input handwritten mathematical expressions, which is more natural and convenient than editors such as Microsoft Equation Editor or LaTeX. Thus, such devices have been widely adopted in various environments, such as offices and educational institutions, especially during the outbreak of COVID-19. This phenomenon has necessitated the development of an accurate model for recognizing HMEs.

The handwritten mathematical expression recognition (HMER) problem differs from the traditional optical character recognition (OCR) problem due to three main reasons:

- The two-dimensional (2D) nature of HMEs adds additional complexity compared with the one-dimensional OCR problem. This can be demonstrated by the following mean squared error (MSE) function: $\frac{1}{N}\sum_{i=1}^{N}(p_i - t_i)^2$.
- The existence of more than 1,500 unique symbols [BFS17], which are often difficult to distinguish from each other (for example, "O", "o", "0", "$\mathbf{O}$", and "$\mathcal{O}$"), especially when considering the variation in handwriting styles. Combined with its structural nature, this leads to an infinite number of combinations of symbols and spatial relationships in MEs.
- The existence of long-term dependencies and correlations among symbols in MEs. For example, "(" and ")" are used to contain subexpressions, and if those subexpressions contain other long subexpressions, these dependencies are challenging to learn.

On a more personal note, as engineering students, we read, write, and share HMEs on a daily basis. Initially suggested by Haya, we thought that applying

the concepts we have learned in the Deep Learning course to tackle the HMER problem would not only be interesting but also very relevant.

1.2 Project Schedule

The project was executed as per the following timeline:

- **Week 1 / 18-25 March**: Literature review of existing works (Haya Monawwar and Sungjoo Chung)
- **Week 2 / 26 - 1 April**: Pre-processing dataset (Haya Monawwar)
- **Week 3 / 2 - 8 April**: Developing a Baseline Model (Haya Monawwar)
- **Week 4 / 9 - 15 April**: Improvement of the Baseline Model through Hyperparameter Optimization and Fine-Tuning (Haya Monawwar)
- **Week 5 / 16 - 22 April** : Structural Modification of the Baseline Model (Sungjoo Chung)
- **Week 6 / 23 - 29 April** : Finalizing Model Improvement and Conducting Ablation Studies (Sungjoo Chung)
- **Week 7 / 30 - 6 May**: Preparing final results and presentation, submitting the final report, and updating the project GitHub (Haya Monawwar and Sungjoo Chung)

While the tasks mentioned were primarily undertaken by the individual named, both team members contributed collaboratively whenever required. GitHub's project management features were utilized to track progress, assign responsibilities, and mark tasks as complete or pending. A detailed list of tasks is available on the project's GitHub page, and all associated materials—including the proposal, reports, presentation, and final code—can be accessed through the project's Github repository. Fig. 1 illustrates the project's task progress, showing the number of open and completed tasks over the allotted timeline. Out of a total of nine tasks, eight were successfully completed, with only one left unfinished. The incomplete task involved implementing a Transformer-based version of the model. However, due to time and resource constraints, the project's focus was strategically adjusted to a comparative study between our baseline and improved CRNN models. As a result, the Transformer-based approach was considered redundant and subsequently excluded from the final implementation.

2 Problem Definition and Dataset

Since we are engineers at the basic level, we tend to think of project problems that either closely relate to dilemmas that all engineers face and/or are closely related to our PhD research field. One major problem that we faced when taking down handwritten notes is that it is a task and a half to convert them into

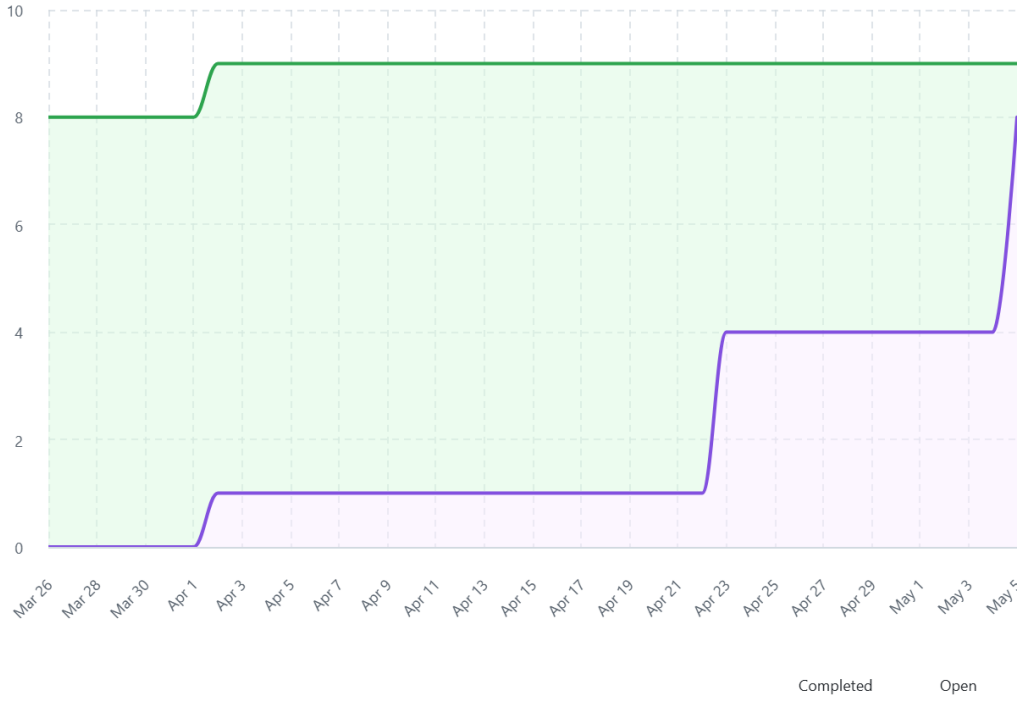


Figure 1: Burn chart depicting this project's progress

LaTex format (a step which most researchers need to perform to type their research publications). Hence, the task at hand is the recognition of mathematical expressions from handwritten data, which involves recognizing symbols and understanding their spatial relationships in a two-dimensional layout. The dataset we used, the CROHME dataset, consists of more than 12,000 examples in image and symbolic format, with each image corresponding to a labeled mathematical expression [Mah+19]. Validation was done for the 2014 and 2019 versions of the dataset, whereas testing was done only for the 2016 version of the dataset.

To process these images, we employed a custom preprocessing pipeline (code shown in preprocessing.py) that imports image files from the dataset. It then normalizes and renders the ink strokes into a canvas suitable for feeding into a neural network, using the image rendering equation below:

$$\text{scale} = \frac{\text{Image size} - (2 \times \text{padding})}{\max(\max x - \min x, \max y - \min y)} \tag{1}$$

This equation scales the coordinates of each stroke to fit within the image dimensions, ensuring that all expressions are normalized for training. $\min_x$ and $\min_y$ represents the minimum values of the x and y coordinates across all

strokes, essentially identifying the leftmost and bottommost points in the raw data; whereas, $\max_x$ and $\max_y$ represent the maximum values of the x and y coordinates, identifying the rightmost and topmost points. The aim of scaling the coordinates was to make sure that they fit within the canvas while preserving the aspect ratio of the formula. Padding was also added to ensure the strokes did not touch the image edges.

The next step was the tokenization of these labelled equations. The labels for the training data were tokenized, meaning that each symbol in a formula was assigned an index in the vocabulary. This is crucial for converting the text labels into a format that the neural network can understand. The output of this step was processed images and their corresponding labels, which were saved in directories for training, validation, and testing. The script also generated a vocabulary file containing the index mappings for each symbol. Inspiration for this part of the implementation was taken from [Ngu24].

3 Model Development and Training Pipeline

After preprocessing, we moved on to the development of recognition models. Initially, there were plans to experiment with Transformer-based models (TrOCR), but due to the time constraints and complexity of the models, this idea was discarded in favor of a more tractable approach using CRNNs. The intermediate step here was the creation of a custom dataset class (`CROHMEDataset`) that was implemented to load images and their corresponding labels from the dataset. The labels were tokenized using the `Vocab` class, converting each symbol into an integer index. Here, data augmentation was also performed:

- First, the image is resized to a fixed size of 100×100, ensuring consistent dimensions across all images.
- A small random rotation is applied, allowing the model to become more robust to minor rotational variations in the input data.
- Slight variations in lighting conditions were introduced to randomly adjust the brightness and contrast of the image, enhancing the model's ability to generalize to different lighting scenarios.
- Next, the image is converted to a PyTorch tensor using `transforms.ToTensor()`, which scales the pixel values from the range [0, 255] to [0, 1].
- Finally, the image is normalized by subtracting the mean (0.5) and dividing by the standard deviation (0.5), helping to stabilize and speed up the training process by ensuring the data is centered around zero.

This combination of transformations not only prepares the images for model input but also introduces variability to improve generalization during training. With the preprocessed and augmented dataset ready, we proceeded to build a baseline model to evaluate initial performance.

3.1 Baseline CRNN Model

The baseline model consisted of Convolutional Neural Networks (CNNs) for feature extraction, followed by a Bi-directional LSTM (BiLSTM) to capture the sequential dependencies of the mathematical symbols. In the subsequent iteration, we enhanced the CRNN model by:

1. Using a pretrained ResNet18 model for better feature extraction.
2. Freezing the early layers of the ResNet to reduce computation time.
3. Adding Batch Normalization and Dropout layers for regularization and preventing overfitting.

This iterative refinement process aimed to improve the model's ability to accurately recognize complex expressions while ensuring faster convergence during training. The training of both models involved iterative improvements based on performance on the validation set.

3.2 Model Training and Evaluation

3.2.1 Training Process

The training pipeline was set up using PyTorch, with the following components:

- **Dataset and Data Loader:** The custom `CROHMEDataset` class was implemented to load images and their corresponding tokenized labels from the dataset (to convert each symbol into an integer index). The data loader used a custom HMERDataset class built on tf.keras.utils.Sequence to support CTC-based training for handwritten expression recognition. The HMERDataset is a custom Keras generator that loads batches of preprocessed grayscale images and tokenized labels, pads label sequences, and computes input and label lengths for CTC loss. The data loader reads image paths and space-separated labels from text files, preprocesses each image to grayscale (128×32), normalizes it, and encodes labels into numeric sequences using a provided vocabulary. Labels are padded to equal lengths, and both input and label lengths are computed for CTC loss compatibility.
- **Model Architecture:** The model was built using a pretrained ResNet18 backbone for feature extraction, followed by a BiLSTM for sequence modeling. A Connectionist Temporal Classification (CTC) loss function was used to handle the misalignment between input images and output labels.

3.2.2 Iterative Training with Epochs

Training involved multiple epochs, with each epoch computing the CTC loss. CTC loss was found to be particularly suitable for this problem because in handwritten mathematical expressions, the positions of symbols in a sequence

are highly variable, and we do not have direct alignment between the input image and the output sequence of characters or symbols.

CTC loss works by predicting a probability distribution for each frame in the sequence and aligning these predictions with the target sequence of labels using the best possible alignment. Mathematical Expression Recognition is a sequence-to-sequence problem, where the input image is a sequence of pixels (processed by a CNN), and the output is a sequence of symbols (recognized by the LSTM). The sequence is highly variable due to different handwriting styles, sizes of characters, and the complexity of mathematical notation (e.g., exponents, roots, fractions). CTC loss facilitates this alignment in sequence prediction without requiring a one-to-one alignment between input and output [Gra+06].

The optimizer used was Adam with a learning rate of 1×10^{-4}, and learning rate scheduling was applied using `ReduceLROnPlateau` to adjust the learning rate based on the validation loss. During each epoch, the model's training and validation performances were evaluated using the Word Error Rate (WER) and Character Error Rate (CER), which were computed using the Levenshtein distance.

WER is a common metric used in tasks that involve transcribing text, such as in Optical Character Recognition (OCR). It measures the performance of a system by comparing the predicted sequence of words or symbols against the ground truth. WER is particularly relevant for recognizing mathematical expressions because these expressions are composed of multiple symbols that together form meaningful terms (words in OCR or formulae in this case).

WER is computed using the Levenshtein distance (edit distance) between the predicted and true word sequences using the equation below:

$$\text{WER} = \frac{S + D + I}{N}$$

Where S is the number of substitutions, D is the number of deletions, I is the number of insertions, and N is the number of words in the ground truth [HTB20].

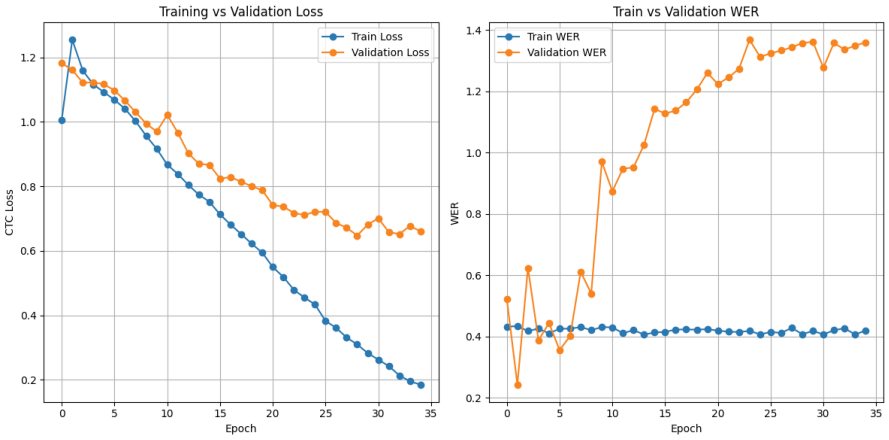
While WER focuses on word-level errors, CER is more granular, calculating the character-level error rate. CER is particularly useful when evaluating the fine-grained accuracy of systems that deal with symbol recognition, such as recognizing individual symbols in mathematical formulas. CER is computed by comparing the predicted sequence of characters with the ground truth at the character level.

This is crucial because mathematical expressions often contain symbols that must be identified accurately. Small differences in symbol recognition (e.g., "1" vs. "l" or "x" vs. "X") can result in significant changes to the meaning of the expression. CER is computed by simply obtaining the ratio between the number of character-level errors and the total number of characters.

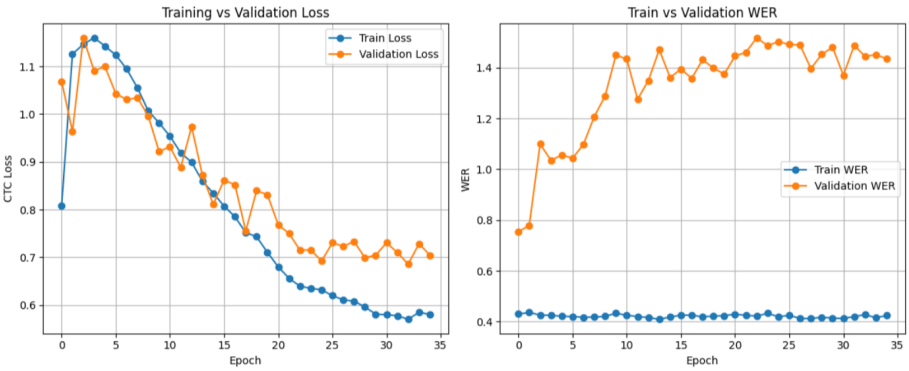
At the end of each epoch, the validation loss and error rates were tracked, and the model's performance was plotted.

3.3 Training Results

The model's performance was evaluated across several epochs, as shown in the plots below, which depict the training and validation CTC loss and WER (Word Error Rate) over the course of 20 epochs. Due to unsatisfactory results, it was increased to 35 epochs. The results from both iterations can be seen in Fig. 2a and Fig. 2b.



(a) Baseline Model Training and Validation Progress - Iteration 1



(b) Baseline Model Training and Validation Progress - Iteration 2

Figure 2: Baseline Model Training and Validation Progress

3.3.1 Training vs Validation Loss (CTC Loss)

The first plots in Fig. 2a and Fig. 2b illustrate the CTC loss during training and validation. Initially, the training loss decreased significantly, indicating that the model was learning to recognize patterns in the mathematical expressions.

However, the validation loss began to diverge from the training loss after a few epochs, suggesting that the model started to overfit to the training data, as evidenced by the increasing gap between the two curves. This is a common issue when the model performs well on the training set but struggles to generalize to unseen data.

3.3.2 Train vs Validation WER (Word Error Rate)

The second plots in Fig. 2a and Fig. 2b show the WER during training and validation. Early in the training process, the training WER was consistently low, indicating that the model was accurately recognizing the mathematical expressions in the training set. However, the validation WER remained relatively high and fluctuated significantly, which suggests that the model was struggling with generalizing its learning to the validation data. This high WER in the validation set further supports the idea that the model is overfitting.

From both trials, it can be seen that the training process showed a good initial learning phase with quick improvements in training performance, but the validation performance highlighted the need for better generalization. Unfortunately, my results did not improve more than 40% WER even after fine-tuning the model with additional strategies to reduce this gap, such as early stopping and learning rate decay.

4 Improving the Baseline Model

Even after hyperparameter optimization and fine-tuning, the baseline model's WER did not exceed 40%. This signified that a more fundamental approach had to be taken to exceed the 40% WER threshold for the model to produce meaningful predictions.

It was at this stage that we decided to follow the CRNN architecture proposed in [SBY16]. In the baseline model, the output of the CNN's final layer is passed through an adaptive average 2D pooling function, which flattens the feature map into a vector. This vector is then used as input to the RNN, which aims to capture the sequential patterns present in mathematical expressions (MEs). However, MEs often contain vertically stacked elements such as fractions or superscripts, and flattening these spatial features can lead to the loss of important structural information.

To address this problem, [SBY16] proposed to extract a sequence of feature vectors from the feature maps produced by the CNN at the final layer, which is the input for the RNN. This means that the i-th feature vector is produced by concatenating the i-th columns of all the feature maps at the final layer. A detailed depiction of the CRNN framework depicted in [SBY16] and adopted in our work can be seen in Fig. 3.

Once the feature sequences are extracted from the CNN, a bi-directional LSTM is implemented to capture the contextual dependencies within the sequence. This architecture is especially effective for the HMER task, as math-

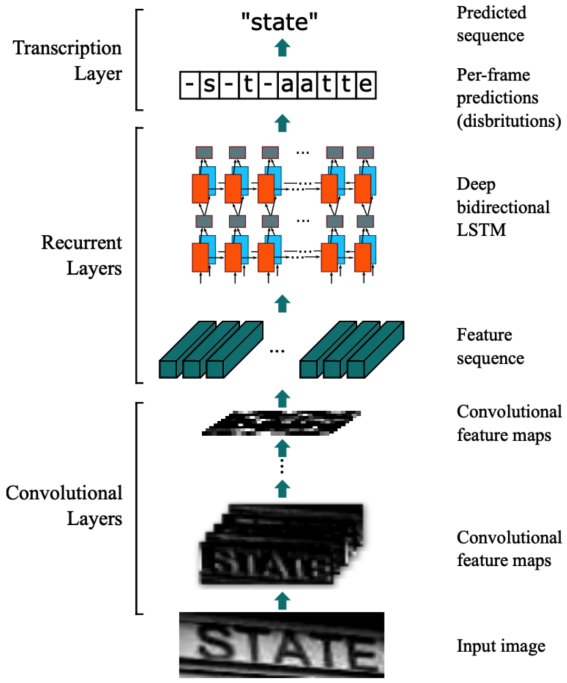


Figure 3: Architecture of CRNN [SBY16]

ematical expressions often contain symbols such as "(" and ")" that exhibit long-term dependencies in both forward and backward directions. Combined with the spatial features extracted from the CNN, we expected this architecture to yield significantly improved results than the baseline model. The final transcription layer decodes the output of the LSTM based on the tokens of mathematical symbols contained in the training and validation datasets.

Apart from the abovementioned modifications, all other methods, including pre-processing and training pipeline, were kept consistent for subsequent iterations of model improvements.

4.1 Initial Results

Apart from adjusting the architecture, an additional change was made regarding the image size. While [SBY16] uses variable widths and only fixes the height of the images to 32, wechose to fix the image sizes to 32×300, in order to accelerate the training process as it enabled faster batch processing by avoiding dynamic padding. The same augmentation made in the baseline model was applied to improve the model's generalizability.

Preliminary results showed very promising results, yielding WER of 19.98%

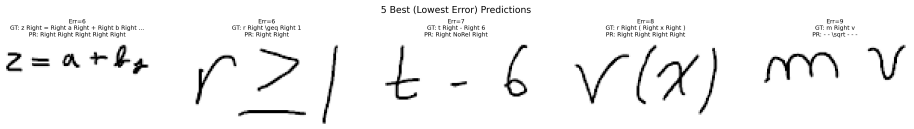


Figure 4: 5 Best Predictions of the Proposed Model in its First Iteration

and 21.66% on the training and validation sets, respectively. However, after obtaining the predictions of the model, it was evident that the model's improvement was not due to it making more accurate predictions, but due to it dominantly predicting the most frequently occurring token. As can be seen in Fig. 4, the predictions of the model are dominated by structural tokens such as "Right". Structural tokens were part of the labels for the HME images in order to provide contextual dependencies to the model.

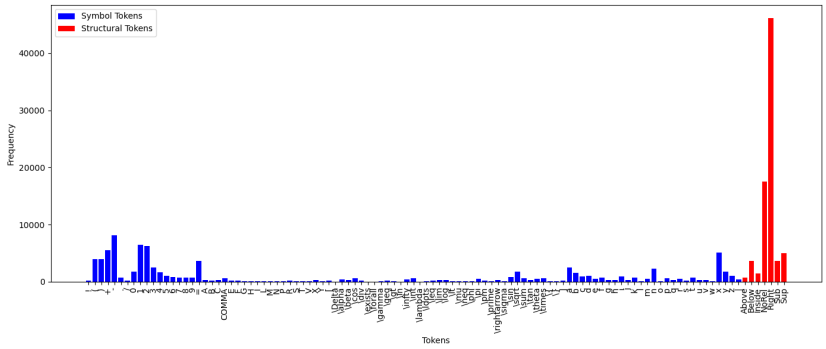
However, it seems like rather than using this contextual dependency to make accurate predictions, the model opted to dominantly output predictions of these structural tokens, thereby inflating the accuracy.

A further investigation into the frequency of structural tokens and symbol tokens revealed that the frequency of structural tokens was similar to the frequency of symbol tokens. However, there are only 7 structural tokens (Right, Above, Below, Inside, Sub, Sup, NoRel), while there are 101 symbol tokens, meaning that the individual frequencies of structural tokens were significantly higher than those of symbol tokens. Fig. 5a illustrates that the structural token "Right" is the most frequently occurring token in the label of the training dataset, appearing a total of $46,147$ times, which constitutes roughly 30% of all token frequency. From Fig. 5b, we can see that even the second most frequent structural token (NoRel) appears roughly twice as many times as the most frequent symbol token ($-$). Thus, the model's behavior can be attributed to the huge imbalance in token frequencies.

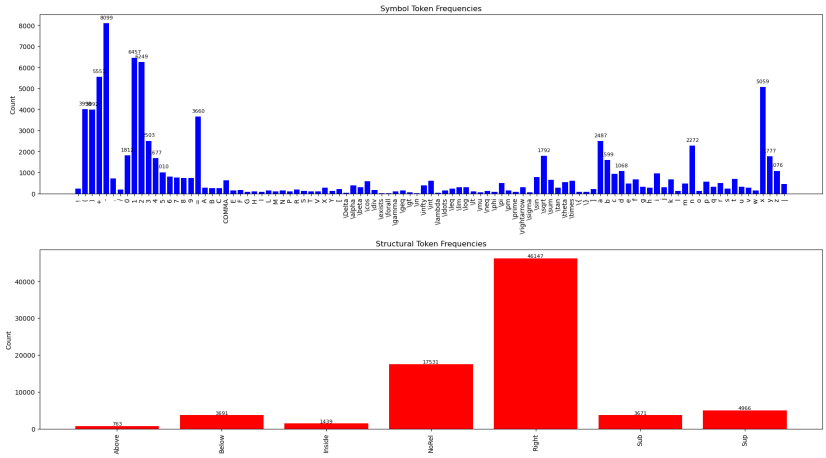
Modifications were made to address this problem, such as penalizing the predictions of structural tokens by incorporating a regularization term in the loss function or implementing a higher class weight for mathematical symbol tokens. However, these efforts proved ineffective. Then, the structural tokens were removed altogether, but this approach yielded a significantly higher WER of 43.29% and 37.14% during training the validation, respectively.

4.2 Changing the Decoding Strategy

After observing that the model's predictions were dominated by structural tokens, it became evident that this behavior stemmed not only from the imbalance in token frequencies but also from the decoding strategy. All previous iterations of the model are trained using the Connectionist Temporal Classification (CTC) loss, which is alignment-free as it enables the model to predict sequences



(a) Frequency of Tokens



(b) Comparison of Symbol and Structural Tokens

Figure 5: Frequency analysis of tokens: (a) token-level frequency, (b) category-wise comparison.

of tokens with intermediate blank labels and repeated symbols. While CTC is powerful for handling variable-length outputs without frame-level alignment, it does not constrain the model to output the most semantically coherent sequence. It only optimizes the probability of the correct label sequence over all valid alignments.

As a result, decoding becomes a critical post-processing step for translating the model's raw outputs into a final sequence of tokens. All the previous iterations of the model rely on greedy decoding: the most likely token at each timestep is selected independently. However, while simple and efficient, this strategy does not incorporate the contextual dependencies inherent within MEs, thus leading to erroneous predictions, such as the phenomenon shown in Section 4.1, from the previous iterations.

To address this, wereplaced greedy decoding with beam search decoding, which considers multiple candidate sequences at each timestep and maintains the top-k most likely sequences (with beam width set to 10 in our experiments). This allows the decoder to explore a broader solution space, shifting its focus from predicting the most likely token at each time step to predicting the most likely sequences of tokens. This change was made on top of removing the structural tokens in the token vocabulary list.

The improvement yielded from this change was significant, especially in terms of the predictions that the model was generating. Furthermore, unlike the previous iterations, we see no signs of overfitting to the training data, as the loss during training and validation plateaus around 20 epochs.

We can observe from Fig. 6 that the CTC loss and WER of the model are higher than the previous iterations. While this would generally mean that the model's performance is worse, a closer look at the predictions generated by the model tells a different story.

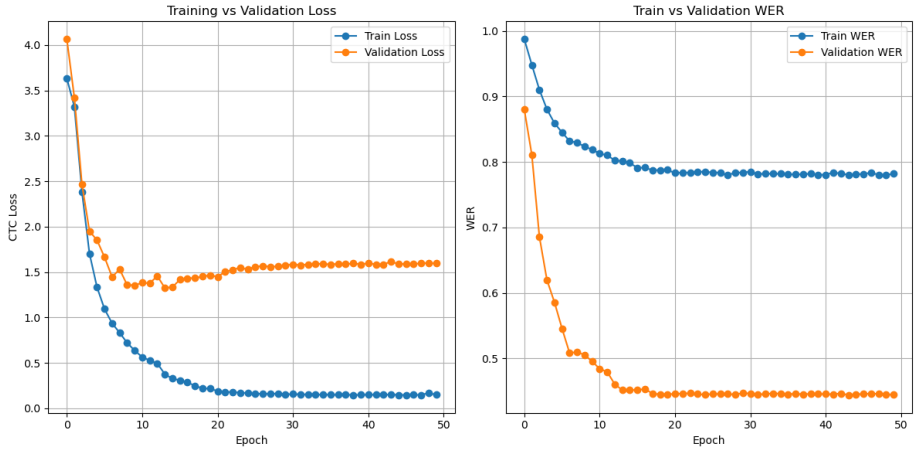


Figure 6: Loss and WER Progression for Training and Validation Dataset

Fig. 7 illustrates the model's predictions across a range of error levels. Compared to the outputs shown in Fig. 4, the improvement is evident. The model achieves high accuracy on shorter mathematical expressions, although its performance tends to degrade as the expressions grow longer and more structurally complex.

Some of the model's errors are, in fact, understandable. For instance, in the 7^{th} image (counting from left to right), the model misclassifies the digit "9" as a "q", incorrectly predicting the label $\sqrt{98}$ as $\sqrt{q8}$. In the 18^{th} image, it misinterprets the symbol "C" as a left parenthesis, resulting in the prediction $(\times($ instead of the correct $C \times C$. While such expressions are clearly invalid to a human reader, the visual similarity between certain handwritten symbols makes these errors plausible from the model's perspective.

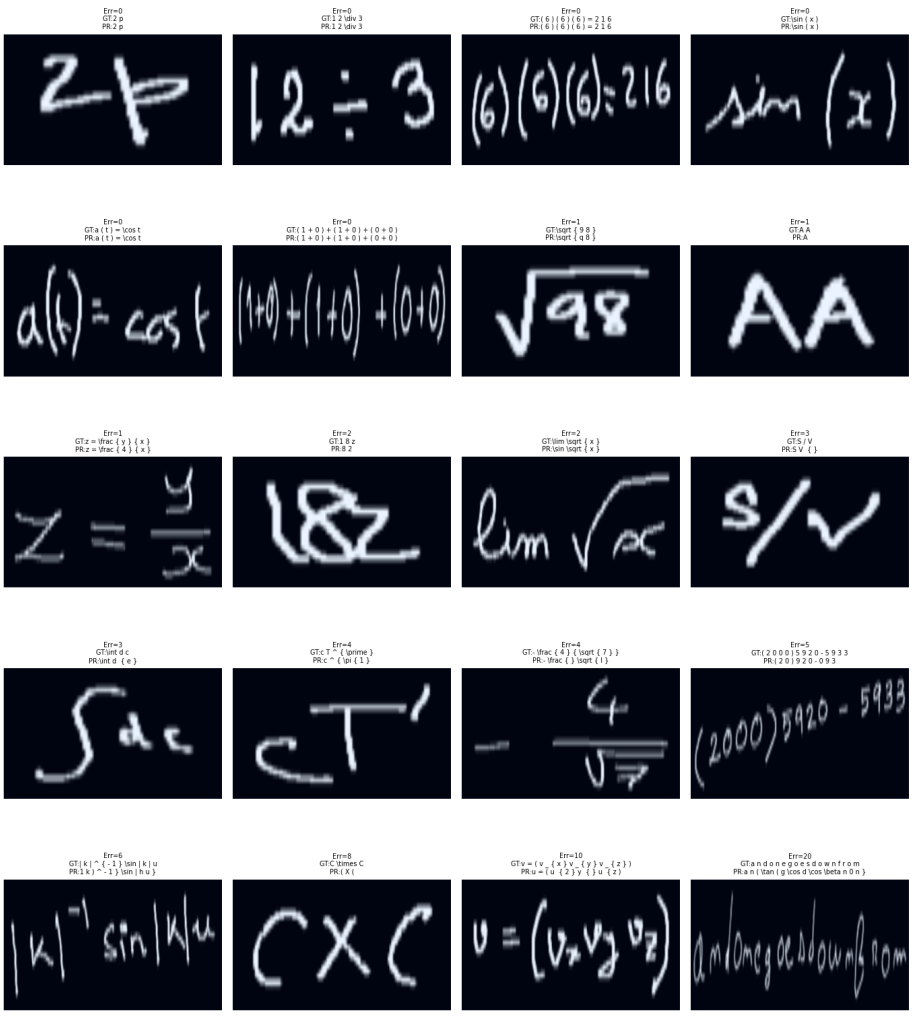


Figure 7: Prediction Samples across a Range of Error Values

5 Post-Processing

During the post-processing stage, we noticed that the predictions made by the model contained the token | between most valid tokens (note that the predictions listed in Fig. 7 omits these tokens for better illustration of the improvement of the model, while the CTC loss and WER in Fig. 6 are calculated without omitting these tokens) as illustrated in Table 1. This phenomenon happens because the CTC loss function relies on blank tokens to enable alignment flexibility in its raw output. While greedy decoding collapses blank tokens and consecutive duplicates automatically, beam search decoding does not inherently remove these

blank tokens as it outputs the raw token paths, therefore inflating the WER and consequently underestimating the accuracy of the model.

Ground Truth	Prediction	Error
kN	k | N	1
12	1 | 2	1
Pa	p | { | a | }	6
19	7 | 9	2
26	2 | 6	1
1m	1 | m	1
Nm	N	1
Hz	H | { | z | }	5
kg	k	1
mv	m | v	1

Table 1: Examples of predictions containing inserted blank tokens (|) and the corresponding Levenshtein error counts.

A simple post-processing step was conducted by removing all | tokens that were inserted as a blank token, and the model's performance was re-evaluated based on the post-processed outputs. Furthermore, the predictions were categorized into two groups: short sequences ($\leq$ 14 tokens) and long sequences ($\geq$ 14 tokens) to emphasize the limitation of the CTC loss function on long sequences.

Expression Length	Original WER	Cleaned WER
Short ($\leq$ 14 tokens)	0.7639	0.2385
Long ($>$ 14 tokens)	0.5344	0.3936

Table 2: Comparison of original and cleaned Word Error Rate (WER) for short and long expressions.

Table 2 summarizes the true performance of the model. It can be seen that the WER improves by roughly 3 times for the short sequences after post-processing, suggesting that the model's performance has significantly improved after implementing all the aforementioned modifications.

6 Final Model

Based on the insights gained through architectural changes, decoding strategies, and post-processing, we consolidated the most effective components into a final model. The finalized architecture and training setup are summarized below in Table 3. These hyperparameters were obtained through iterative experimentation and empirical tuning. Multiple configurations were tested, and the selected

settings provided the best trade-off between training stability, accuracy, and generalization to complex handwritten mathematical expressions.

In the final model, the CNN backbone was switched from ResNet18 to VGG16. VGG's simple and uniform architecture preserves spatial resolution better than ResNet, based on our experimentation, which can be attributed to its consistent convolutional structure helping to capture local spatial relationships more effectively, which is an essential property for accurately interpreting spatially organized expressions in HMER.

Table 3: Final Model Architecture and Training Parameters

Component	**Hyperparameter**	**Value / Setting**
Input	Image Size	(3, 32, 300)
	Batch Size	32
CNN Backbone	Architecture	VGG16
RNN	Type	2-layer Bidirectional LSTM
	Hidden Size	256
	Dropout (between layers)	0.5
Fully Connected Layer	Input / Output Size	512 (256×2) / 110 tokens
	Dropout	0.3
Loss	Loss Function	CTC
Optimizer	Type	Adam
	Learning Rate	5e-4
Scheduler	Type	ReduceLROnPlateau
	Factor and Patience	0.5 (halves LR on plateau), 2
Decoding	Beam Width	10
Training	Epochs	50

Summary and Conclusion

This project successfully explored a CRNN-based approach for Handwritten Mathematical Expression Recognition (HMER), aiming to convert ink-based mathematical expressions into LaTeX format. The methodology included preprocessing the CROHME 2019 dataset by rendering and normalizing image inputs and tokenizing symbolic labels with a custom vocabulary class. A Convolutional Recurrent Neural Network (CRNN) architecture was employed, leveraging a pretrained ResNet18 for feature extraction and Bi-directional LSTMs (BiLSTM) for sequence modeling. The Connectionist Temporal Classification (CTC) loss function was used to align the variable-length input-output sequences without requiring explicit alignment.

Initial results showed strong training performance with reduced CTC loss; however, increased validation loss and fluctuating Word Error Rates (WER) revealed issues of overfitting and poor generalization. To improve the baseline model, we introduced structural modifications, enhanced the decoding strategy, and incorporated post-processing techniques. Specifically, the feature sequence

extraction method proposed in [SBY16] was adopted to preserve spatial relationships within mathematical expressions, and the greedy decoding strategy was replaced with beam search to produce more semantically coherent predictions.

A key insight was that raw beam search outputs often included CTC blank tokens, such as the "|" symbol, which inflated WER. A post-processing step was implemented to remove these tokens, resulting in a significant performance improvement—reducing WER from over 40% to approximately 23.85% for short expressions. Notably, this not only improved quantitative performance but also yielded predictions that were more coherent and semantically meaningful.

Throughout the project, we learned the critical importance of decoding strategies and post-processing in CTC-based models, particularly for structurally complex tasks like HMER. Our work demonstrates that even lightweight CRNN-based models can achieve strong performance when paired with careful architectural design and thoughtful post-processing. Future improvements could explore explicit alignment supervision, attention mechanisms, or encoder-decoder frameworks to better capture long-range dependencies and nested expression structures. Additionally, this model could be integrated into real-time applications to enable on-the-fly conversion of handwritten equations into LaTeX format.

Appendix: Code Listings

The full code implementation of this project can be found on GitHub.

Code Snippets from `preprocessing.py`

Listing 1: Preprocessing Pipeline

```
def normalize_and_render(traces, trace_groups=None, img_size=256, padding=10):
...
    all_points = np.array(all_points)
    min_x, min_y = np.min(all_points, axis=0)
    max_x, max_y = np.max(all_points, axis=0)
    scale = (img_size - 2 * padding) / max(max_x - min_x, max_y - min_y + 1e-6)
    canvas = np.ones((img_size, img_size), dtype=np.uint8) * 255
    for t_id in keys:
        if t_id in traces:
            trace = np.array(traces[t_id])
            trace -= [min_x, min_y]
            trace *= scale
            trace += padding
            trace = trace.astype(np.int32)
            for i in range(1, len(trace)):
                pt1 = tuple(trace[i - 1])
```

```
                pt2 = tuple(trace[i])
                cv2.line(canvas, pt1, pt2, color=0,
                    thickness=2)
    return canvas
```

Listing 2: Custom Dataset and Vocab Class

```
class Vocab(object):
    def __init__(self, vocab_file):
        self.word2index = {}
        self.index2word = {}
        with open(vocab_file, 'r') as f:
            for i, line in enumerate(f):
                word = line.strip()
                self.word2index[word] = i
                self.index2word[i] = word
        self.word2index['<blank>'] = len(self.word2index)
        self.index2word[self.word2index['<blank>']] = '<
            blank>'

class CROHMEDataset(Dataset):
    def __init__(self, img_dir, label_file, vocab, transform
        =None):
        self.img_dir = img_dir
        self.transform = transform
        self.vocab = vocab
        self.samples = []
        with open(label_file, 'r', encoding='utf-8') as f:
            for line in f:
                parts = line.strip().split('\t')
                if len(parts) != 2:
                    print(f"Skipping malformed line: {line.
                        strip()}")
                    # Attempt to delete corresponding image
                        if found
                    if parts and parts[0].endswith('.png'):
                        image_path = os.path.join(self.
                            img_dir, parts[0])
                        if os.path.exists(image_path):
                            os.remove(image_path)
                            print(f"Deleted image: {
                                image_path}")
                    continue

                img_name, label = parts
                self.samples.append((img_name, label))
    def __len__(self):
        return len(self.samples)

    def __getitem__(self, idx):
```

```
        img_name, label = self.samples[idx]
        img_name = os.path.basename(img_name)
        img_name = os.path.splitext(img_name)[0] + '.png'
        img_path = os.path.join(self.img_dir, img_name)

        try:
            image = Image.open(img_path).convert("RGB")
            if self.transform:
                image = self.transform(image)
            label_indices = [self.vocab.word2index[word] for
                word in label.split()]
            return image, torch.tensor(label_indices, dtype=
                torch.long)
        except Exception as e:
            print(f"Skipping {img_name}: {e}")
            return None
```

Code Snippets from CRNN_HMER.py

Listing 3: Evaluation Metrics

```
def levenshtein_distance(a, b):
    m = len(a) + 1
    n = len(b) + 1
    dp = np.zeros((m, n))

    for i in range(m):
        dp[i][0] = i
    for j in range(n):
        dp[0][j] = j

    for i in range(1, m):
        for j in range(1, n):
            cost = 0 if a[i - 1] == b[j - 1] else 1
            dp[i][j] = min(dp[i - 1][j] + 1, dp[i][j - 1] +
                1, dp[i - 1][j - 1] + cost)

    return dp[m - 1][n - 1]


# Function to calculate Word Error Rate (WER)
def compute_wer(predictions, labels):
    total_words = sum(len(label.split()) for label in labels
        )
    errors = 0
    for pred, label in zip(predictions, labels):
        errors += levenshtein_distance(pred.split(), label.
            split())  # Calculate WER at word level
    return errors / total_words if total_words > 0 else 0
```

```python


# Function to calculate Character Error Rate (CER)
def compute_cer(predictions, labels):
    total_chars = sum(len(label) for label in labels)
    errors = 0
    for pred, label in zip(predictions, labels):
        errors += levenshtein_distance(list(pred), list(
            label))  # Calculate CER at character level
    return errors / total_chars if total_chars > 0 else 0
```

References

[Gra+06] Alex Graves et al. "Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks". In: *Proceedings of the 23rd International Conference on Machine Learning*. ICML '06. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 2006, pp. 369–376. ISBN: 1595933832. DOI: 10.1145/1143844.1143891. URL: https://doi.org/10.1145/1143844.1143891.

[SBY16] Baoguang Shi, Xiang Bai, and Cong Yao. "An End-to-End Trainable Neural Network for Image-Based Sequence Recognition and Its Application to Scene Text Recognition". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39.11 (2016), pp. 2298–2304.

[BFS17] Barbara Beeton, Asmus Freytag, and Murray Sargent. *Unicode support for mathematics*. The Unicode Consortium, Mountain View, CA, USA, Tech. Rep. 25. May 2017.

[Mah+19] Mahshad Mahdavi et al. "ICDAR 2019 CROHME + TFD: Competition on Recognition of Handwritten Mathematical Expressions and Typeset Formula Detection". In: *2019 International Conference on Document Analysis and Recognition (ICDAR)*. 2019, pp. 1533–1538. DOI: 10.1109/ICDAR.2019.00247.

[HTB20] Fukeng He, Jun Tan, and Ning Bi. "Handwritten Mathematical Expression Recognition: A Survey". In: *Pattern Recognition and Artificial Intelligence*. Ed. by Yue Lu et al. Cham: Springer International Publishing, 2020, pp. 55–66. ISBN: 978-3-030-59830-3.

[Ngu24] Cuong Nguyen. *CROHME_CTC_pytorch*. https://www.kaggle.com/code/ntcuong2103/crohme-ctc-pytorch. Accessed: 2025-05-05. 2024.