COLLEGE OF
ENGINEERING, ARCHITECTURE AND TECHNOLOGY

Deep Learning for Recognizing and Converting Handwritten Scientific Equations into LaTeX

Deep Learning (ECEN 5060)

Haya Monawwar and Sungjoo Chung (Group 04)

May 06 2025

Department of Electrical and Computer Engineering
Oklahoma State University – Stillwater Campus

Outline

- Introduction
- Problem
- Dataset
- Methodology
- Results
- Limitations
- Conclusion
- Future Work

Introduction

- As engineers, we often want to fast-track our handwritten notes into digital format for various purposes. But can we digitalize handwritten equations directly?
- …Hence, the motivation for this project.

What if we could extend the problem and get our hand-written equations converted to LaTex version directly?

Problem

Develop a machine learning based system that accurately recognizes and converts handwritten mathematical expressions/scientific equations into LaTeX format using deep learning techniques.

Dataset

- CROHME 2019 (Competition on Recognition of Handwritten Mathematical Expressions)
- Contains approximately 12,000 HMEs from previous CROHME competitions (2014-2019)
- Format:
  - Greyscale Images: Sized 1000x1000 pixels with 5 pixels of padding.
  - SymLG annotations: SymLG is a structured representation of HMEs that captures both individual symbols and their spatial relationships.
- Annotations are used to create a vocabulary split.
- Train/test/validation split: 8,835 / 2,186 / 1,147 (images and captions each)

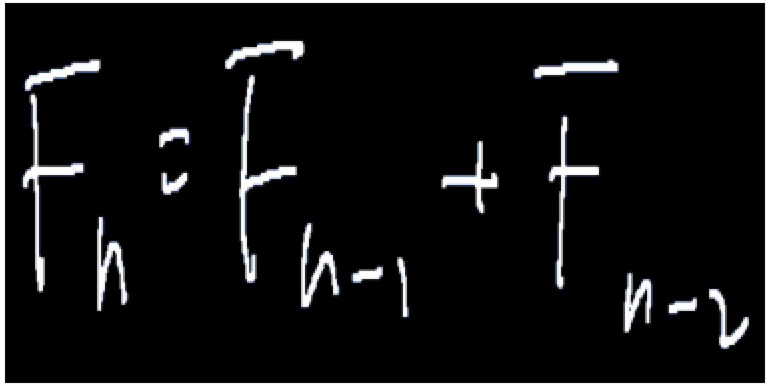
Methodology

Pre-processing

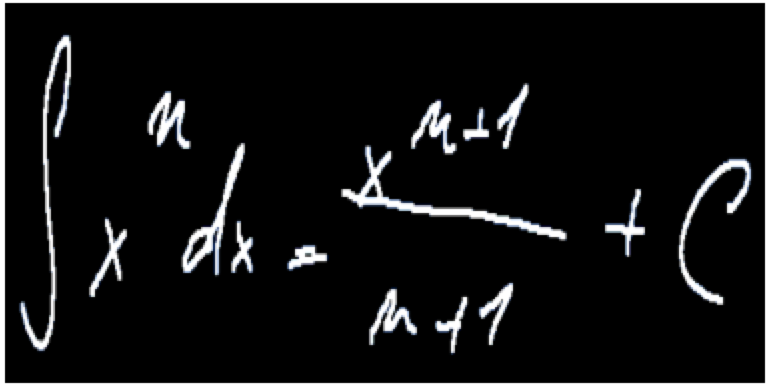
- Tokenizing / creating a unified vocabulary
- Implemented CROHMEDataset to load images and their corresponding tokenized labels.
- Data augmentation:
  - Resizing to 100 x 100
  - Random rotation
  - Color jitter by varying brightness and contrast
  - Normalizing with mean = standard deviation = 0.5
  - Converted images to PyTorch tensors and normalized to [0,1].

Methodology

GT: F _ { n } = F _ { n - 1 } + F _ { n - 2 }

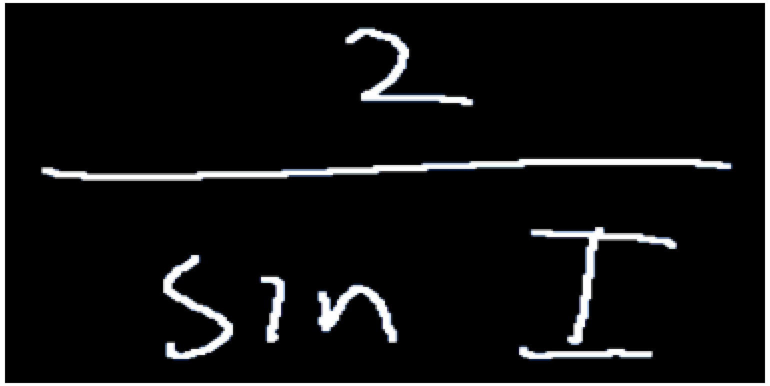


GT: \int x ^ { n } d x = \frac { x ^ { n + 1 } } { n + 1 } + C

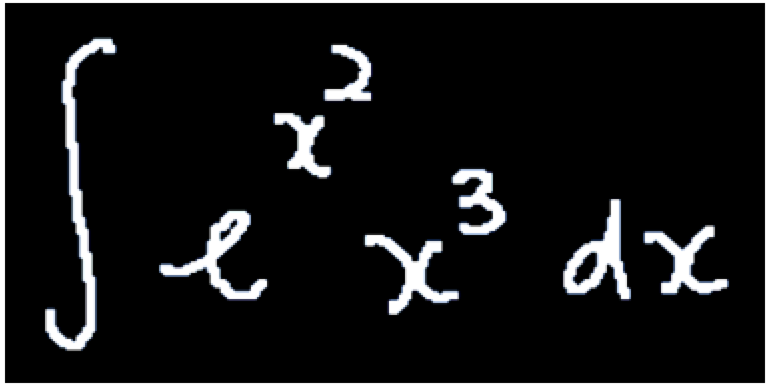


GT: c _ { t }

GT: \frac { 2 } { \sin I }



GT: \int e ^ { x ^ { 2 } } x ^ { 3 } d x



GT: ([R _ { y }])

Methodology

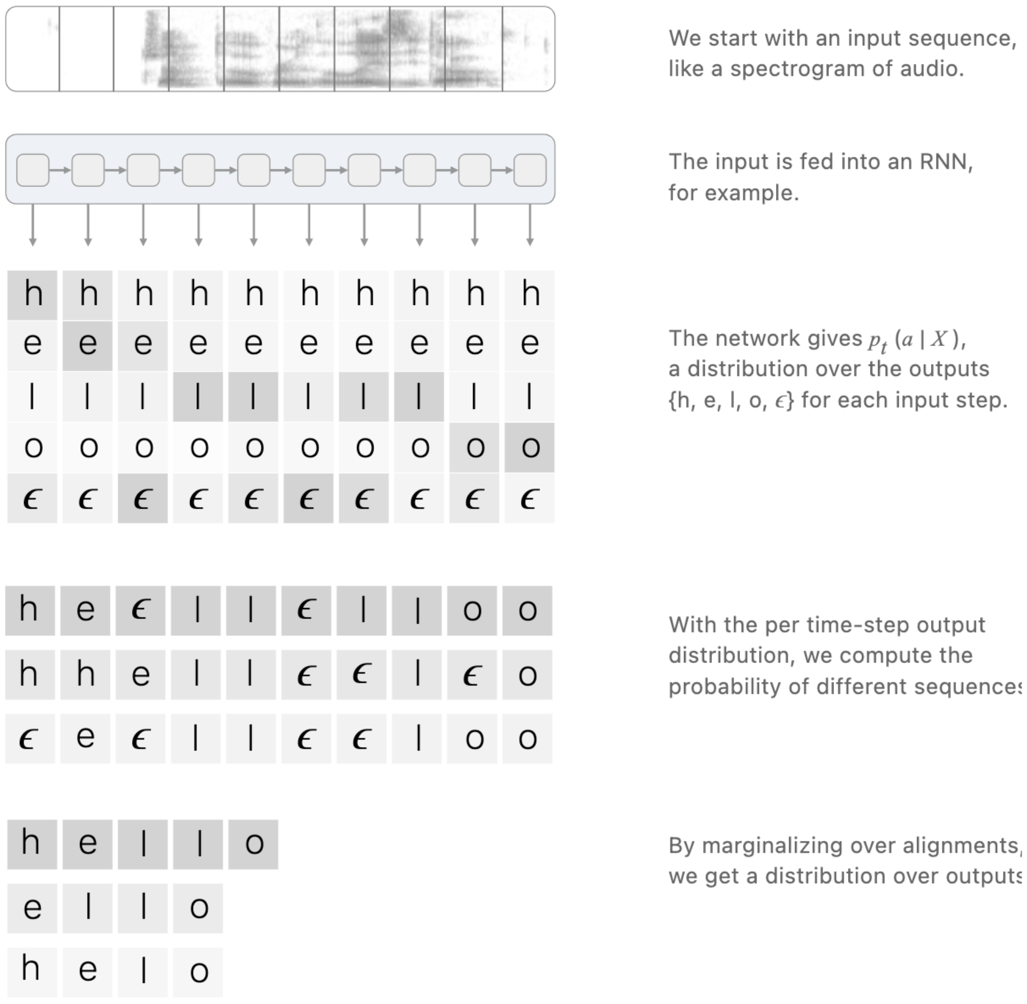
Baseline Model

- Using a CNN-RNN
- Pre-trained ResNet18
- Freezing the early layers of the ResNet to reduce computation time.
- Adding Batch Normalization and Dropout layers for regularization and preventing overfitting.
- Enhancing the LSTM layers to improve temporal sequence handling.
- Uses vocabulary like ‘Right’, ‘Left’, ‘Above’, etc.

```python
def __init__(self, num_classes):
    super(CRNN, self).__init__()
    resnet = models.resnet18(pretrained=False)
    state_dict = torch.load("/kaggle/input/resnet18/resnet18-f37072fd.pth")
    resnet.load_state_dict(state_dict)

    # Freeze first few layers to avoid overfitting
    for param in list(resnet.children())[:5]:
        for p in param.parameters():
            p.requires_grad = False

    # Extract CNN layers up to layer3
    self.cnn = nn.Sequential(
        *list(resnet.children())[:-3],                 # Keep until layer3
        nn.BatchNorm2d(256),                           # Add BatchNorm after last conv layer
        nn.Dropout2d(p=0.3)                            # Dropout after batchnorm
    )

    # Bidirectional LSTM with dropout
    self.rnn = nn.LSTM(
        input_size=256,
        hidden_size=256,
        num_layers=2,
        dropout=0.5,              # Dropout between LSTM layers
        bidirectional=True,
        batch_first=True
    )

    self.dropout_fc = nn.Dropout(p=0.3)               # Dropout before final classification
    self.fc = nn.Linear(512, num_classes)

def forward(self, x):
    x = self.cnn(x)  # (B, 256, H, W)
    x = nn.functional.adaptive_avg_pool2d(x, (1, x.size(3)))  # (B, 256, 1, W)
    x = x.squeeze(2)  # (B, 256, W)
    x = x.permute(0, 2, 1)  # (B, W, 256)
    x, _ = self.rnn(x)
    x = self.fc(x)
    return x
```

Loss function

- CTC Loss [2]
  - Enables models to learn alignments between input sequences and output label sequences without requiring explicit frame-level annotations.
  - It allows for flexible spacing and repeated characters, while handling intermediate blank tokens.

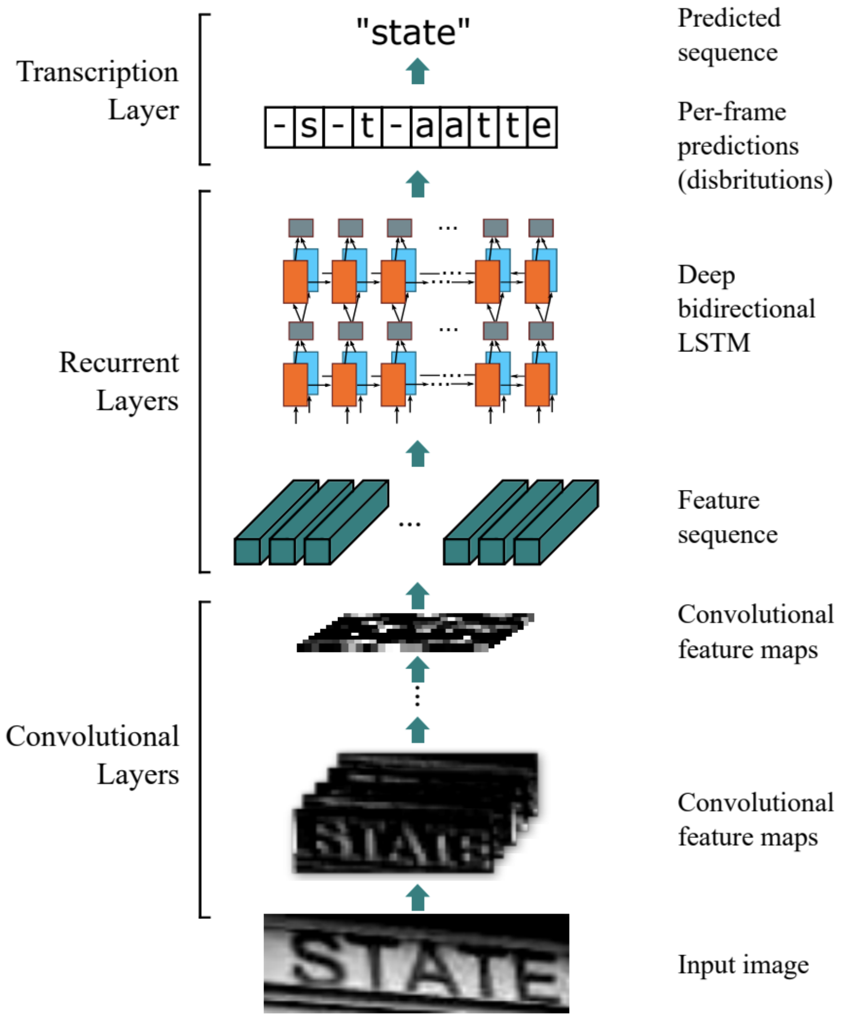


Evaluation Metric

- **Levenshtein distance:** a string metric that measures how many single-character edits are needed to change one string into another. These edits include:
  - Insertion
  - Deletion
  - Substitution
- Character Error Rate (CER): Levenshtein distance / Total number of characters
- Word Error Rate (WER): Levenshtein distance (words) / Total number of words

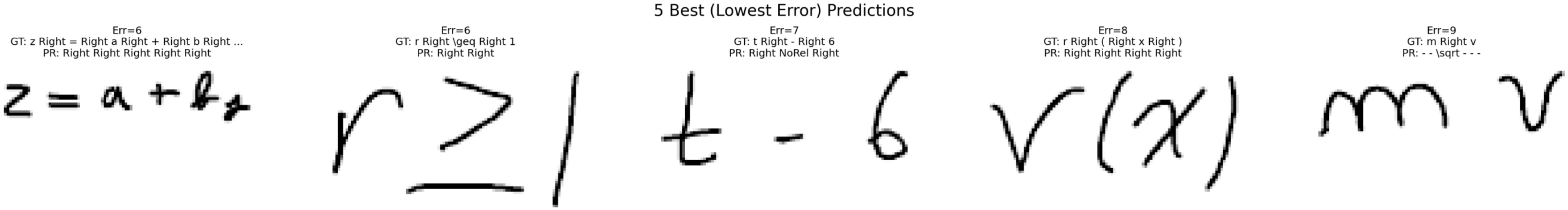
Methodology – 1st iteration

- CRNN framework from [1]
  - Convolution layer: extract features from images
  - Recurrent layer: makes prediction for each frame of the feature sequence (bi-directional)
  - Transcription layer: translates per-frame predictions into labeled sequence
- Feature maps are "sliced" to keep the height, helps distinguish stacked symbols like fractions or superscripts



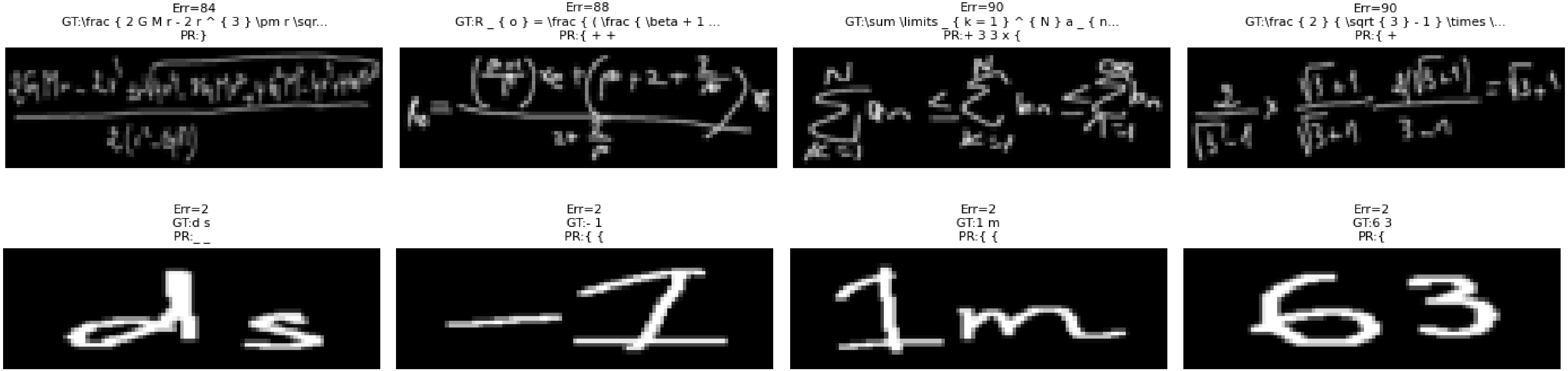
Methodology – 1st iteration results

- Image is resized to 32 x 300 (unlike paper, which uses varying width)
- Train and Validation WER: 19.98% and 21.66%
- However, predictions were spammed with "Right", inflating the accuracy
- Indicators such as "Right", "Below", "Inside"… are unnecessary due to the CRNN framework



Methodology – 2nd iteration

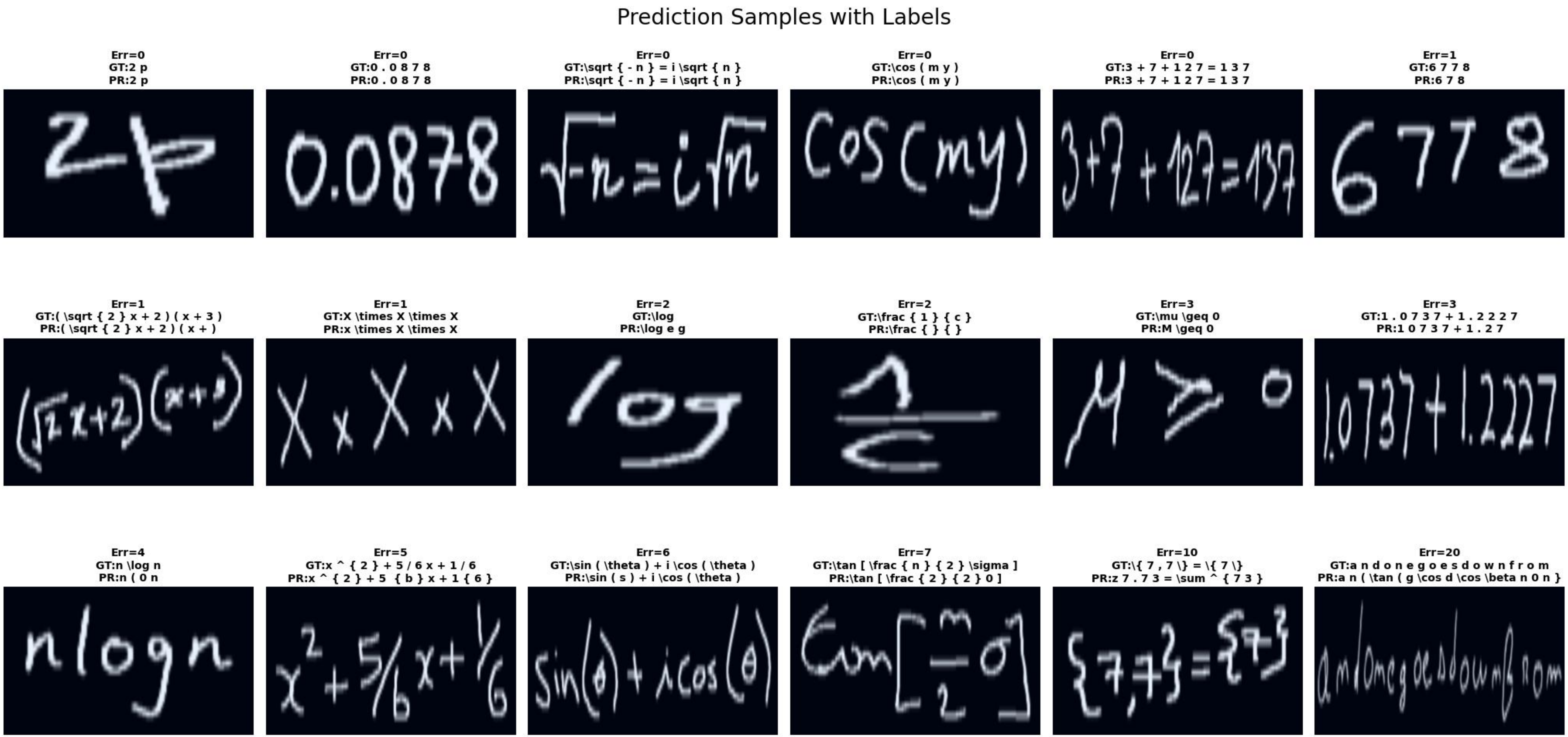
- Modified the vocabulary to exclude structural / relationship indicators
- Model's predictions were still very bad even after hyperparameter optimization (Train and validation WER: 43.29% and 37.14%)



Methodology – 3rd iteration

- Greedy decoding:
  - At each timestep, selects the single most probable token
  - Does not consider the long-term sequence likelihood. A locally optimal token may lead to poor global predictions.
- Beam decoding:
  - Maintains the top-k most likely sequences at each timestep (e.g., top 5 partial predictions).
  - Common math structures (fractions, subscripts, braces) can be preserved due to global consistency.
- Due to the nature of CTC loss, beam decoding is especially effective in the proposed framework

Methodology – 3rd iteration results



Methodology

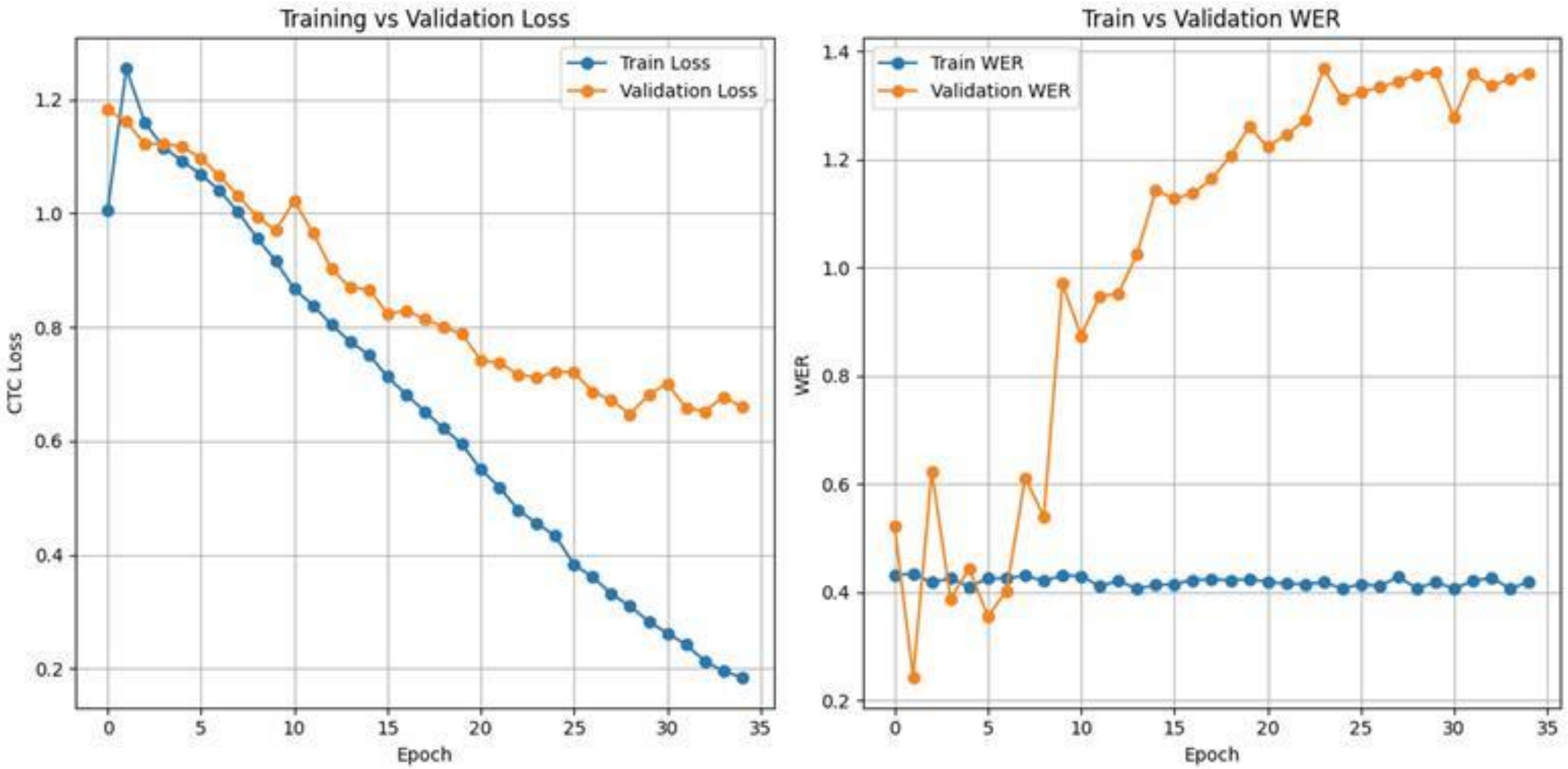
Post-processing

<table>
<tr><th>GroundTruth</th><th>Prediction</th><th>Error</th></tr>
<tr><td>k N</td><td>k | N</td><td>1</td></tr>
<tr><td>1 2</td><td>1 | 2</td><td>1</td></tr>
<tr><td>P a</td><td>p | { | a | }</td><td>6</td></tr>
<tr><td>1 9</td><td>7 | 9</td><td>2</td></tr>
<tr><td>2 6</td><td>2 | 6</td><td>1</td></tr>
<tr><td>1 m</td><td>1 | m</td><td>1</td></tr>
<tr><td>N m</td><td>N</td><td>1</td></tr>
<tr><td>H z</td><td>H | { | z | }</td><td>5</td></tr>
<tr><td>k g</td><td>k</td><td>1</td></tr>
<tr><td>m v</td><td>m | v</td><td>1</td></tr>
</table>

Notice something?

Final Results

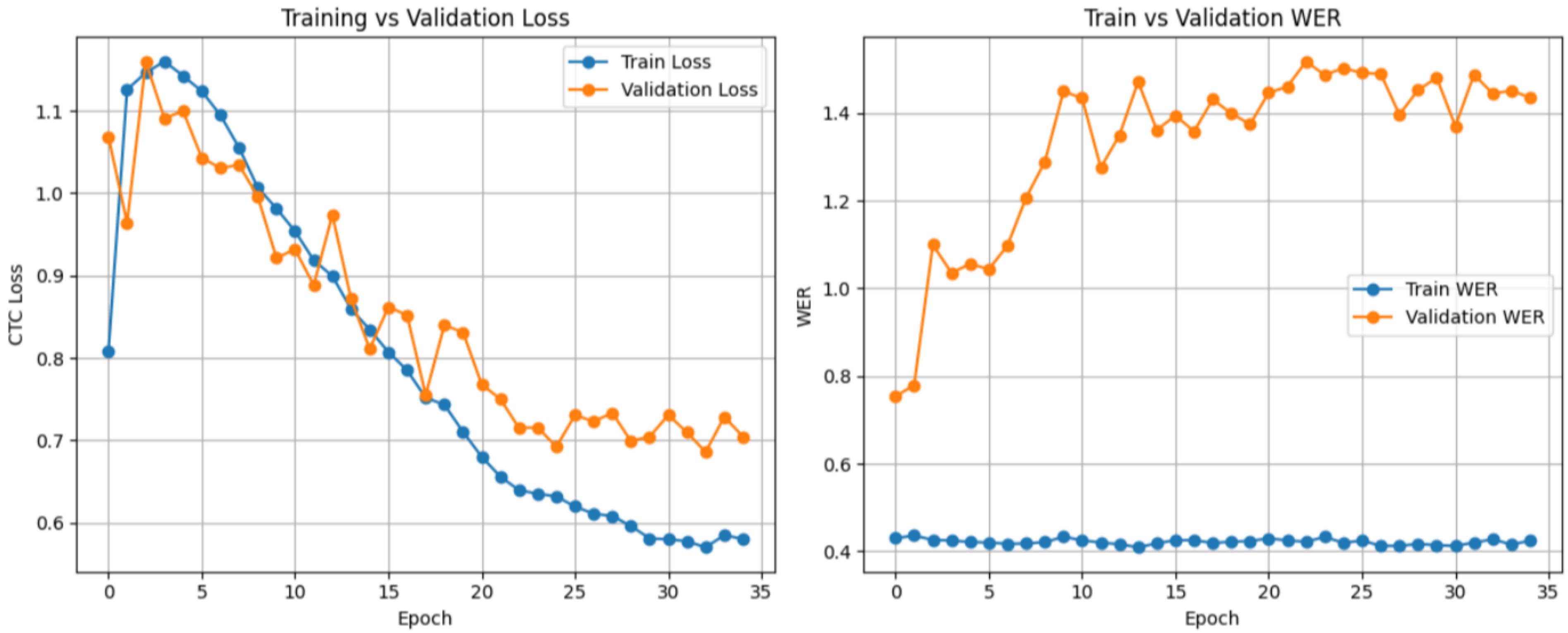
Training loop/Intermediate Results



Baseline model – Iteration 1

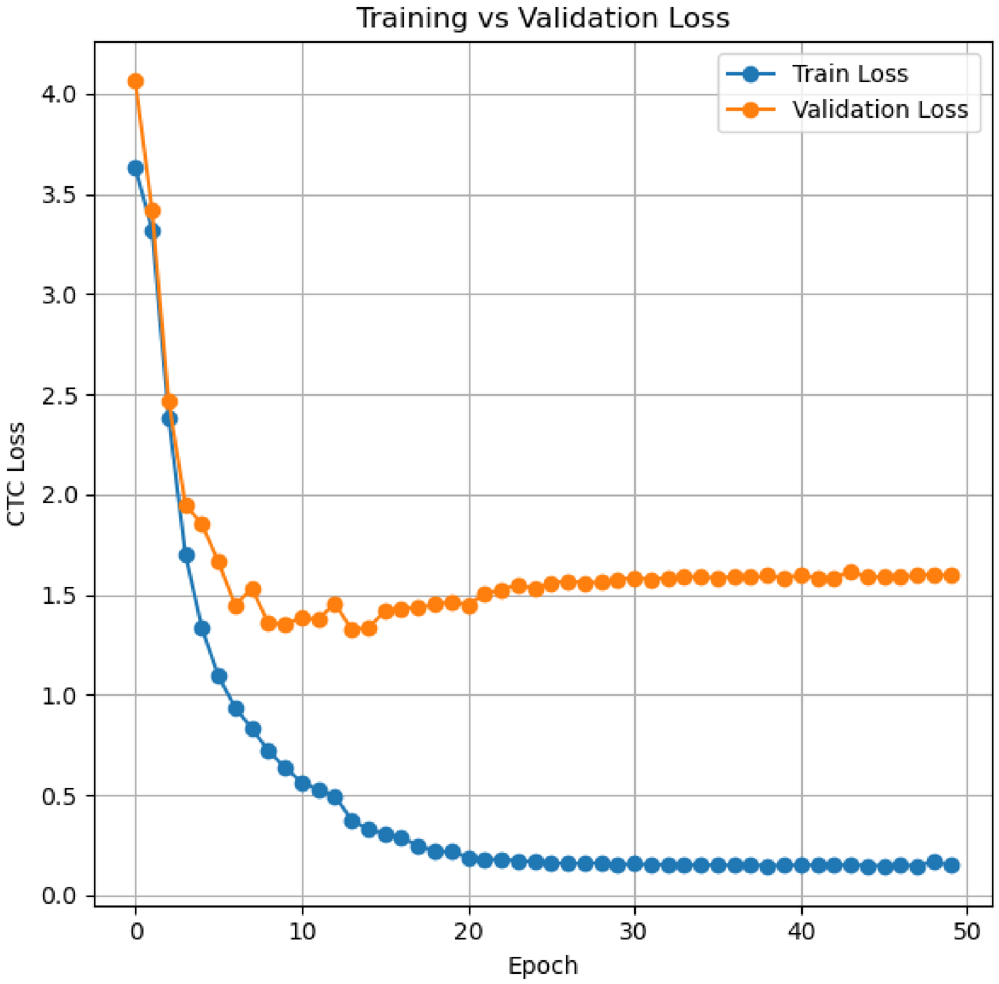
Final Results

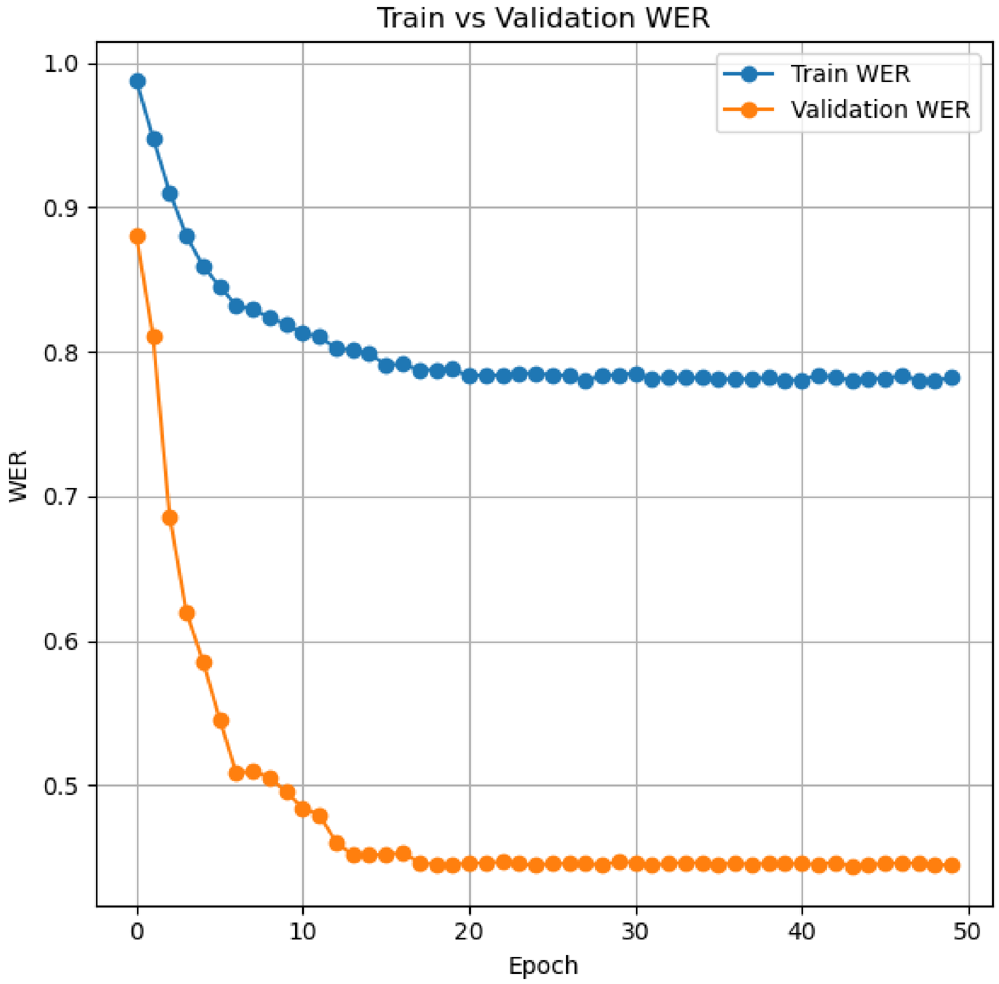
Training loop/Intermediate Results



Baseline model – Iteration 2

Final Results





Final Model Architecture and Parameters

Component	Hyperparameter	Value / Setting
Input	Image Size	(3, 32, 300)
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

Post Processing

- Due to the garbage token insertion, the WER were inflated
- After preprocessing, we split the predictions by length, and compare the original and cleaned WER
  - Split was done by finding the median expression length value
  - We see a significant improvement in overall WER, especially for short expressions

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

Limitations

- CTC loss suffers with long sequences, as evident from WER
- CTC assumes the output sequence is *monotonically aligned* with the input (left-to-right), which doesn't always hold for math expressions
- The model treats output as a flat sequence of tokens, thus it can't handle hierarchical or spatial relationships
- Even with beam search, the model makes decisions at the token level, which means it can still miss globally optimal expression sequences.

Conclusion

- Developed a deep learning pipeline to recognize and convert handwritten mathematical expressions from images into LaTeX format
- Implemented a CRNN-based model with CTC loss for sequence prediction.
- Iteratively refined the system by adjusting model architecture, vocabulary, and decoding strategies (greedy → beam search).
- Integrated post-processing to remove garbage tokens, significantly improving WER rates, especially for shorter sequences

Future Work

- Attempt a Transformer based model – does it out-perform our work?
- Experimenting with deeper trained models.
- Incorporate additional data augmentation strategies, such as elastic distortions or random cropping, to make the model more robust to various writing styles and formats.
- Integrate the proposed mechanism into a mobile/web application to enable wide-spread use.

References

1. Baoguang Shi, Xiang Bai, and Cong Yao. “An End-to-End Train- able 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.
2. Hannun, "Sequence Modeling with CTC", Distill, 2017.

THANK YOU

Questions?