AN INTEGRATED PLATFORM FOR HANDWRITTEN MATHEMATICAL EXPRESSION RECOGNITION

AND ADAPTIVE PROBLEM SOLVING

PROJECT REPORT

Submitted by

KONDI NANDA GOPAL UMESH RAJU (21CS1017)

VENGADESA BOOPATHI P (21CS1058)

GOPESH A (21CS2003)

Under the guidance of

DR. F. SAGAYARAJ FRANCIS

Professor

Department of Computer Science and Engineering

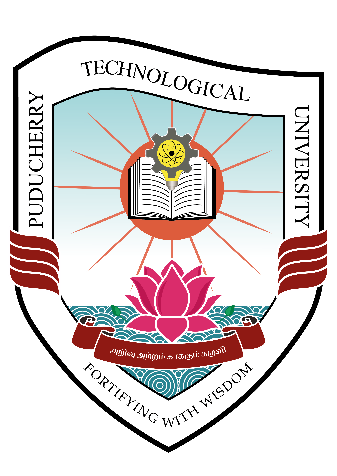
Submitted to the Puducherry Technological University, in partial fulfillment

of the requirement for the award of degree

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PUDUCHERRY TECHNOLOGICAL UNIVERSITY

PUDUCHERRY – 605 014

MAY 2025DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PUDUCHERRY TECHNOLOGICAL UNIVERSITY

PUDUCHERRY – 605 014.

BONAFIDE CERTIFICATE

This is to certify that the Project work titled “AN INTEGRATED PLATFORM FOR HANDWRITTEN MATHEMATICAL EXPRESSION RECOGNITION AND ADAPTIVE PROBLEM SOLVING” is a Bonafide work done by KONDI NANDA GOPAL UMESH RAJU (21CS1017), VENGADESA BOOPATHI P (21CS1058) and GOPESH A (21CS2003) In partial fulfillment for the award of the degree of Bachelor of Technology in Computer Science and Engineering of the Puducherry Technological University and that this work has not been submitted for the award of any other degree of this/any other institution.

|  |  |  |
| --- | --- | --- |
| ((Dr. F. SAGAYARAJ FRANCIS) Professor  Project Guide |  | (Dr. E. ILAVARASAN)  Professor  Head of the Department |

*Submitted for the University Examination held on\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

Internal Examiner External Examiner

**ACKNOWLEDGEMENT**

I am deeply indebted to **Dr. F. SAGAYARAJ FRANCIS,** Professor Department of Computer Science and Engineering, Puducherry Technological University, Puducherry, for his valuable guidance throughout this project.

I also express my heartfelt gratitude to **Dr. ILAVARASAN E,** Professor & Head (CSE) for giving constant motivation in succeeding my goal.

With profoundness I would like to express my sincere thanks to **Dr. S. MOHAN,** Vice-Chancellor, Puducherry Technological University, for his kindness in extending the infrastructural facilities to carry out my project work successfully.

I would be failing in my duty if I do not acknowledge the efforts of the Project Co-Ordinator, **Dr. N. SIVAKUMAR**, Professorand the Project Review Panel members, viz. **Dr. F. SAGAYARAJ FRANCIS,** Professor, **Dr. J. JAYABHARATHY,** Professor,  **Dr. K. SATHIAMURTHY,** Professor, **Dr. N. SIVAKUMAR,** Professorand **Dr. P. SALINI** Associate Professor for shaping our ideas and constructive criticisms during project review.

I also express my thanks to all the **Faculty** and **Technical Staff** members of the CSE department for their timely help and the **Central Library** for facilitating useful reference materials.

I would be failing in my duty if I don’t acknowledge the immense help extended by my friends, who have always been with me in all my trialsand tribulations and encouraging me to complete.

**KONDI NANDA GOPAL UMESH RAJU**

**VENGADESA BOOPATHI P**

**GOPESH A**

### ****ABSTRACT****

### **Handwritten mathematical problem-solving presents two core challenges: accurately digitizing handwritten expressions and computing their solutions across diverse mathematical domains. This project introduces an integrated platform that unifies annotation-free offline Handwritten Mathematical Expression Recognition (HMER) with adaptive problem-solving, automating the workflow from image input to verified solution. The recognition module employs a ResNet-18 encoder to generate hierarchical visual features, which are processed through a self-supervised contrastive patch-learning framework that directly models symbol relationships without clustering or manual preprocessing. Overlapping 16×16 patches are embedded into a discriminative 128D space and structured into a dynamic graph, where edges represent top-2 cosine similarities between patches. A Graph Attention Network (GAT) refines these relationships, enabling a Transformer decoder to generate LaTeX markup with a target Expression Recognition Rate (ExpRate) of 59.57% on the CROHME2019 dataset.**

### **Beyond recognition, the system features a dynamic solver engine that interprets LaTeX output and applies domain-specific algorithms. For algebraic equations, a machine learning classifier selects between symbolic (SymPy) and numeric (SciPy) solvers based on structural features like equation degree and term count. The system also supports trigonometric simplification, matrix operations (via NumPy), and calculus tasks (ODEs/PDEs). This adaptive approach ensures optimal computational strategies for diverse problem types.**

### **Deployed as a scalable web application, the platform combines a React frontend for intuitive image upload and solution visualization with a Flask backend for robust processing. By eliminating manual preprocessing, symbol annotations, and static solvers, the system bridges offline handwriting recognition with intelligent computation, offering students, educators, and researchers a unified tool for accurate transcription and domain-aware problem-solving.**

**TABLE OF CONTENT**

| **CHAPTER NO.** | **TITLE** | **PAGE NO.** |
| --- | --- | --- |
|  | **BONAFIDE CERTIFICATE** | ***ii*** |
|  | **ACKNOWLEDGEMENT** | ***iii*** |
|  | **ABSTRACT** | ***iv*** |
|  | **LIST OF FIGURES** | ***viii*** |
|  | **LIST OF TABLES** | ***ix*** |
|  | **LIST OF ABBREVIATIONS** | ***ix*** |
|  | **LIST OF SYMBOLS** | ***ix*** |
|  |  |  |
| **1** | **INTRODUCTION** | 1 |
|  | 1.1 OVERVIEW | 1 |
|  | 1.2 OBJECTIVE OF STUDY | 1 |
|  | 1.3 MOTIVATION/ NEED FOR THE STUDY | 2 |
|  | 1.4 ORGANIZATION OF THE CHAPTERS | 2 |
|  |  |  |
| **2** | **LITERATURE REVIEW** | 3 |
|  | 2.1 LITERATURE REVIEW BASED ON VARIOUS RESEARCH PAPERS | 3 |
|  | 2.2 SUGGESTIONS BASED ON LITERATURE REVIEW | 5 |
| **3** | **EXISTING WORK** | 6 |
|  | 3.1 OFFLINE HMER WITH GETD | 6 |
|  | 3.2 TECHNIQUES USED IN EXISTING WORK | 6 |
|  | 3.3 EXISTING ARCHITECTURE | 7 |
|  | 3.4 ARCHITECTURE DESCRIPTION | 7 |
|  | 3.4.1. Symbol Detection (Yolov5-Based)  3.4.2. Symbol Graph Construction  3.4.3. Graph Encoder (Gat-Based)  3.4.4. Transformer Decoder | 7  8  8  8 |
|  | 3.5 ALGORITHM | 9 |
|  | 3.6 MODULES USED | 10 |
|  | 3.7 LIMITATIONS OF EXISTING WORK | 10 |
|  | 3.8 RESULT | 11 |
|  |  |  |
| **4** | **PROPOSED WORK** | 12 |
|  | 4.1AN INTEGRATED PLATFORM FOR HANDWRITTEN MATHEMATICAL EXPRESSION RECOGNITION AND ADAPTIVE PROBLEM SOLVING  4.2SELF-SUPERVISED PATCH-BASED REGION PROPOSAL WITH GRAPH NEURAL NETWORK ENCODING FOR HMER | 12  13 |
|  | 4.3 PROPOSED ARCHITECTURE | 13 |
|  | 4.4 ARCHITECTURE OF PROPOSED OFFLINE HMER  4.4.1. Input Preprocessing and Feature Extraction  4.4.2. Self-Supervised Patch Embedding  4.4.3. Adaptive Graph Construction  4.4.4. Graph Attention Network (GAT) Architecture  4.4.5. Transformer Decoder with Cross-Attention  4.4.6. Positional Encoding and Token Embeddings  4.4.7. Autoregressive Training and Inference | 14  14  14  14  15  15  15  16 |
|  | 4.5 MATH SOLVER  4.5.1. Algebraic Equation Solver  4.5.2. Matrix Solver  4.5.3. Trigonometric Equation Solver  4.5.4. Differentiation Solver  4.6 TECHNIQUES USED IN PROPOSED WORK | 16  16  17  18  19  19 |
|  | 4.7 ALGORITHM | 21 |
|  | 4.8 MODULES USED | 22 |
|  | 4.9 RESULT | 23 |
|  |  |  |
| **5** | **SIMULATION RESULTS/ EXPERIMENTAL RESULTS (Existing Work & Proposed Work)** | 24 |
|  | 5.1 DATASET DESCRIPTION | 24 |
|  | 5.2 PERFORMANCE METRICS | 25 |
|  | 5.2.1. Expression Recognition Rate (ExpRate)  5.2.2. Structural Similarity Rate (StruRate)  5.2.3. Top-k Accuracy (≤1, ≤2)  5.2.4. Loss  5.3 EXPERIMENTAL SETUP | 25  25  26  26  26 |
|  | 5.3.1 Hardware Configuration | 26 |
|  | 5.3.2 Software and Libraries | 27 |
|  | 5.4 RESULT | 27 |
| **6** | 5.5 GRAPH  5.6 INFERENCES  5.7 IMPLEMENTATION RESULTS OF EXISTING AND PROPOSED WORK RESULTS WITH SNAPSHOTS  5.7.1 Implementation Results of Existing Work- Offline HMER Module  5.7.2 Implementation Results Proposed Work- Offline HMER Module  5.7.3 Implementation Results of Proposed Work -Math Solver  **CONCLUSION & FUTURE ENHANCEMENTS**  6.1 Conclusion  6.2 Future Enhancements  **REFERENCES** | 28  32  33  33  37  41  45  45  46  48 |

**LIST OF FIGURES**

|  |  |  |
| --- | --- | --- |
| **Figure No.** | **Title** | **Page No.** |
| 3.1 | Existing architecture for HMER | 8 |
| 3.2 | Image To LaTex Conversion with GETD | 12 |
| 4.1 | End to End Pipeline For HMER and Adaptive Solving | 13 |
| 4.2 | Proposed Architecture for HMER | 14 |
| 4.3 | Image to LaTex Conversion | 24 |
| 5.1 | HMER with Algebra Equation Solver | 29 |
| 5.2 | HMER with Matrix Solver | 29 |
| 5.3 | HMER with Trigonometry Equation Solver | 30 |
| 5.4 | HMER with Differentiation Solver | 30 |
| 5.5 | CROHME2014 Loss vs. Epoch | 31 |
| 5.6 | CROHME2014 Accuracy vs Epoch | 32 |
| 5.7 | CROHME2016 Loss vs. Epoch | 32 |
| 5.8 | CROHME2016 Accuracy vs Epoch | 33 |
| 5.9 | CROHME2019 Loss vs. Epoch | 33 |
| 5.10 | CROHME2019 Accuracy vs Epoch | 34 |
| Snapshot 1 | Implementation of existing work OFFLINE HMER | 40 |
| Snapshot 2 | Implementation of proposed work OFFLINE HMER | 45 |

**LIST OF ABBREVIATION**

|  |  |
| --- | --- |
| **Abbreviation** | **Meaning** |
| HMER | Handwritten Mathematical Expression Recognition |
| CNN | Convolutional Neural Network |
| MLP | Multi-Layer Perceptron |
| GAT | Graph Attention Network |
| NT-Xent | Normalized Temperature-scaled Cross-Entropy |
| ROIAlign | Region of Interest Alignment |
| FP16 | 16-bit Floating-Point Precision |
| SymPy | Symbolic Python Library |
| SciPy | Scientific Python Library |
| ODE | Ordinary Differential Equation |
| PDE | Partial Differential Equation |
| API | Application Programming Interface |

**LIST OF SYSMBOLS**

|  |  |
| --- | --- |
| **Symbol** | **Description** |
| x, y | Variables in mathematical expressions |
| \tau | Temperature hyperparameter in contrastive loss |
| z\_i, z\_j | Patch embedding vectors |
| N | Number of patches (nodes) |
| d\_{\text{model}} | Transformer model dimension |
| H, W | Height and width of feature map |
| \mathbf{e} | Edge feature vector in graph |
| \mathrm{ExpRate} | Expression Recognition Rate |
| \oplus | Vector concatenation |

**CHAPTER I**

**INTRODUCTION**

* 1. **OVERVIEW**

The proposed project delivers an end-to-end automated platform for offline handwritten mathematical expression recognition and adaptive problem-solving, addressing the disconnect between existing recognition tools and computational engines. The pipeline begins with grayscale normalization and ResNet-18-based feature extraction, bypassing error-prone steps like binarization or skeletonization [[1].](#ref1) A contrastive region proposal network processes 16×16 overlapping patches, learning discriminative embeddings through self-supervised training without symbol-level annotations. These patches form nodes in a dynamically constructed graph, where edges encode top-2 cosine similarities to model spatial and semantic relationships. A Graph Attention Network (GAT) refines node features through multi-head attention, capturing both local symbol details and global structural context [[1]](#ref1) . Finally, a Transformer decoder autoregressively generates LaTeX sequences, which are parsed and routed to an adaptive solver engine.

The solver employs machine learning to dynamically select between symbolic (SymPy) and numeric (SciPy) methods based on equation complexity, ensuring optimal performance for algebra, trigonometry, matrix operations, and calculus [[11]](#ref6) . Deployed via a React frontend and Flask backend, the platform provides real-time interaction, enabling users to upload handwritten equations and receive step-by-step solutions. This integrated approach eliminates manual intervention, reduces computational overhead, and enhances accessibility for educational and research applications.

* 1. **OBJECTIVE OF THE STUDY**

To develop a self-supervised HMER framework that processes handwritten images through ResNet-based feature extraction, contrastive patch embedding, and dynamic graph construction, eliminating dependency on symbol-level annotations or manual preprocessing.

. To design an adaptive mathematical solver that dynamically selects between symbolic (SymPy) and numeric (SciPy) techniques using ML-based classification, tailored for algebra, trigonometry, calculus, and matrix operations.

To integrate the recognition and solving modules into a scalable web platform using React for frontend interactivity and Flask for backend processing, enabling real-time, user-friendly access to transcription and computational solutions.

**1.3 MOTIVATION/ NEED FOR THE STUDY**

There is a growing need for intelligent systems that bridge handwritten input with computational solvers, especially in academic and research environments where handwritten equations are still dominant. Manual transcription is error-prone and inefficient, while most digital solvers require syntactically correct, typed input. Moreover, existing systems lack adaptability across diverse mathematical domains. This study addresses these gaps by introducing a flexible, accurate, and self-learning platform that democratizes access to advanced mathematical tools through AI, enhancing learning, evaluation, and productivity.

**1.4 ORGANIZATION OF THE CHAPTERS**

The report consists of the following chapters:

Chapter 1 – Introduction

Chapter 2 – Literature Review

Chapter 3 – Existing Work

Chapter 4 – Proposed Work

Chapter 5 – Simulation Results/Experimental Results

Chapter 6 – Conclusion and Future Work

**CHAPTER II**

**LITERATURE REVIEW**

**2.1 LITERATURE REVIEW BASED ON VARIOUS RESEARCH PAPERS**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| S.No. | Title | Journal / Conference (Year) | Proposed Techniques & Concepts | Limitations |
| 1 | Offline Handwritten Mathematical Expression Recognition with Graph Encoder and Transformer Decoder | Elsevier (2023) | Uses a graph attention network to model pairwise spatial relations among symbols and a Transformer decoder to output LaTeX sequences in one end-to-end architecture. | High computation time on large graphs |
| 2 | Stroke Extraction for Offline Handwritten Mathematical Expression Recognition | IEEE Transactions on Pattern Analysis and Machine Intelligence (2021) | Applies skeletonization and morphological tracing to recover stroke order, then incorporates stroke sequence into layout parsing for better symbol segmentation. | Sensitive to noise and low resolution |
| 3 | A Comparison Study on Optical Character Recognition Models in Mathematical Equations and in Any Language | Elsevier(2025) | Benchmarks traditional OCR engines against deep-learning models on math and multilingual text. | Results tied to chosen evaluation datasets |
| 4 | Watch, Attend and Parse: An End-to-End Neural Network Based Approach to Handwritten Mathematical Expression Recognition | Pattern Recognition (2017) | Treats recognition as sequence generation via an attention-based encoder-decoder that learns to focus on symbol regions without explicit segmentation. | Attention may miss fine spatial details |
| 5 | Improving Handwritten Mathematical ExpressionRecognition via Similar Symbol Distinguishing | IEEE Transactions On Multimedia, Vol. 26, 2024 | Introduces hierarchical attention that first attends broadly then refines to finer details, improving handling of dense or overlapping symbols. | Inference speed reduced by two-stage attention |
| 6 | Handwritten Mathematical Expression Recognition using Graph Convolutional Network | ICPR (International Conference on Pattern Recognition) (2022) | Represents each detected symbol as a graph node with edges encoding 2D relationships, then applies a graph convolutional network | Depends on accurate initial symbol detection |
| 7 | SymPy: Symbolic Computing in Python | PeerJ Computer Science (2017) | Provides a pure-Python symbolic mathematics library supporting algebra, calculus, matrices and LaTeX parsing for backend problem solving. | Slow on deeply nested or very large expressions |
| 8 | Symbolic Integration with SymPy: Algorithms and Applications | ACM Communications in Computer Algebra (2021) | Details SymPy’s rule-based rewriting and recursive algorithms for symbolic integration, demonstrating effectiveness on complex integrals. | Rule set does not cover all integral types |
| 9 | Hybrid Symbolic-Numeric Methods for Solving Algebraic Equations | Journal of Symbolic Computation (2012) | Combines symbolic simplification for linear parts with numeric solvers for non-linear components to balance precision and speed. | Numeric phase may introduce rounding errors |
| 10 | Dynamic Approach Selection for Solving Mathematical Problems: Symbolic or Numeric | IEEE Symposium Series on Computational Intelligence (2022) | Employs a machine-learning classifier to analyze features of a problem and dynamically choose between symbolic or numeric solution paths | Classifier misroutes can degrade performance |

### 2.2 SUGGESTIONS BASED ON LITERATURE REVIEW

### The reviewed literature consistently demonstrates that explicitly encoding the spatial and structural relationships between handwritten mathematical symbols substantially enhances recognition performance. Approaches employing graph-based models, such as Graph Attention Networks (GAT) and Graph Convolutional Networks (GCN) [[1]](#ref1), are effective in modeling the two-dimensional arrangement of symbols, including critical constructs like superscripts, subscripts, fractions, and nested expressions. Representing expressions as graphs rather than linear sequences preserves spatial context, which is essential for resolving ambiguities and improving the semantic understanding of complex handwritten math [[3].](#ref3)

### Furthermore, attention mechanisms have proven to be highly effective for both symbol detection and sequence generation in handwritten mathematical expression recognition. Hierarchical attention models, such as coarse-to-fine attention [[4]](#ref4), enable the network to focus progressively from global structural regions down to finer symbol details. Transformer-based decoders utilizing multi-head self-attention further improve the generation of LaTeX sequences by capturing long-range dependencies and maintaining contextual coherence [[3]](#ref3). Incorporating these advanced attention strategies contributes significantly to the robustness and accuracy of recognition systems, especially when dealing with long or structurally complex expressions.

### In addition to recognition, the integration of symbolic computation frameworks, such as SymPy, into the post-processing pipeline is recommended to enhance overall system capabilities [[11]](#ref6). Symbolic and hybrid symbolic-numeric solvers enable the validation, simplification, and solving of recognized expressions, thereby extending the system’s utility in educational, scientific, and engineering applications. Modular architectures that facilitate seamless interaction between recognition and adaptive solving components are therefore advocated, ensuring that the system can handle a broad spectrum of mathematical tasks with improved reliability and accuracy.

### .

**CHAPTER III**

**EXISTING WORK**

**3.1 OFFLINE HANDWRITTEN MATHEMATICAL EXPRESSION RECOGNITION WITH GRAPH ENCODER AND TRANSFORMER DECODER**

The recognition of offline handwritten mathematical expressions (HMER) presents significant challenges due to the inherent structural complexity of mathematical notation and the variability in human handwriting styles. Traditional methods in this domain typically rely on three sequential steps: symbol detection or segmentation, individual symbol recognition, and structural parsing of their 2D relationships [[2]](#ref2). However, these pipelines are often error-prone due to ambiguous symbol boundaries and fail to capture global contextual relationships. While grammar-based techniques offer structural interpretability, they are highly reliant on hand-crafted rules and are not robust to stylistic variations across handwriting [[3]](#ref3).

To overcome these limitations, recent advances in deep learning have led to the development of end-to-end encoder–decoder models that treat HMER as an image-to-sequence generation problem. However, while these methods have shown improved performance, they often lack interpretability because they do not explicitly segment symbols. The research addressed in this paper proposes a novel **Graph-Encoder-Transformer-Decoder (GETD)** framework [[1]](#ref1). It explicitly detects symbols as nodes and encodes their relationships as edges in a graph structure. A Graph Neural Network (GNN) aggregates spatial information, and a Transformer decoder produces the final LaTeX sequence [[1]](#ref1)[[3]](#ref3). This hybrid approach enhances interpretability, boosts recognition accuracy, and allows for semi-supervised training to alleviate the need for exhaustive symbol-level annotations.

**3.2 TECHNIQUES USED IN EXISTING WORK**

**Graph-Based Symbol Representation**

Existing models represent handwritten equations as graphs, where each node corresponds to a detected symbol and edges represent spatial relationships based on Line-of-Sight (LOS). This allows capturing structural layout explicitly..

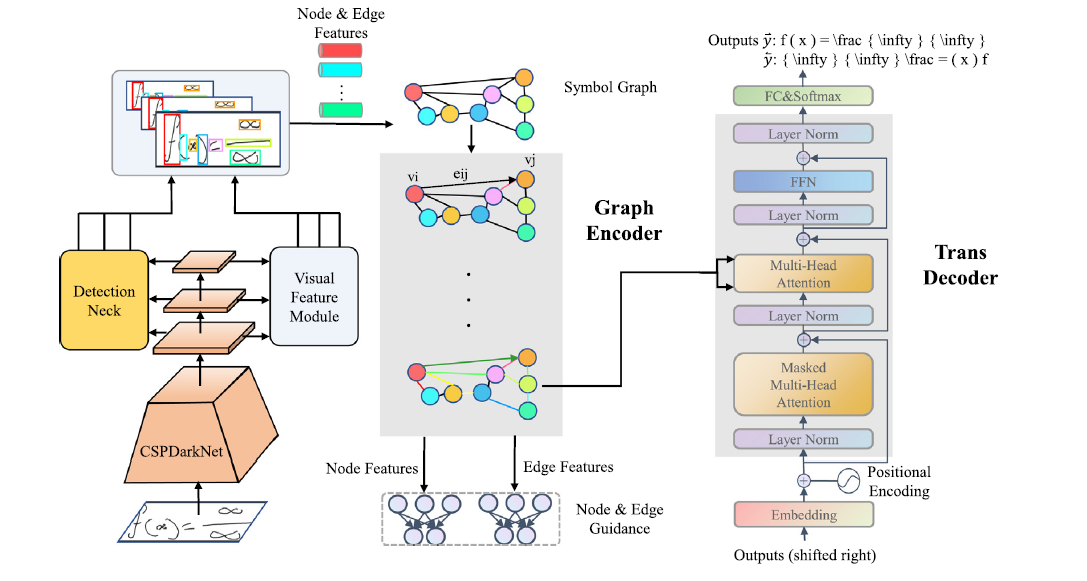
**Graph Attention Network (GAT) Encoding**

The models use GATs to propagate contextual information across the graph. This allows the network to understand the local and global structure of mathematical expressions, improving symbol relation modelling [[1]](#ref1).

**Transformer Decoder with Bidirectional Training**

Instead of using traditional RNNs, a Transformer decoder is employed for sequence generation. The decoder uses bidirectional training (left-to-right and right-to-left), which enhances robustness and output balance in LaTeX sequence generation [[1]](#ref1).

**3.3 EXISTING ARCHITECTURE**



**Figure 3.1: Existing architecture for HMER**

**3.4 ARCHITECTURE DESCRIPTION**

The GETD model comprises three core modules: symbol detection, graph encoder, and Transformer decoder. Below is a detailed breakdown:

**3.4.1. Symbol Detection (YOLOv5-Based)**

Objective: Detect and localize individual symbols (e.g., digits, operators) in the input image.

Backbone: CSPDarknet53 (Cross-Stage Partial Network) extracts multi-scale features.

Loss Functions:

Box Regression: Complete-IoU (CIoU) Loss for precise localization

Objectness Loss: Binary cross-entropy (BCE) to distinguish symbols from background.

**3.4.2. Symbol Graph Construction**

Nodes: Detected symbol regions (bounding boxes).

Edges: Defined via line-of-sight (LOS) relationships. Two symbols are connected if no third symbol occludes their direct spatial line.

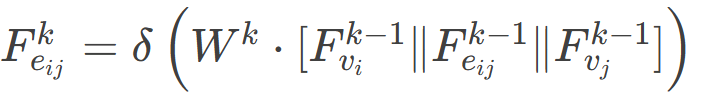
Feature Extraction:

Node Features: ROIAlign extracts visual features from symbol regions, followed by dimensionality reduction (FC layer).

Edge Features: ROIAlign on the union of two connected symbols, then compressed via FC layer.

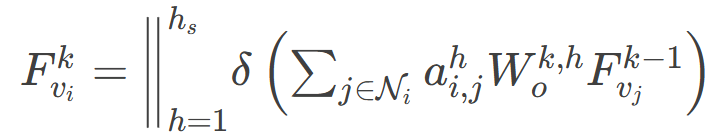
**3.4.3. Graph Encoder (GAT-Based)**

Graph Attention Network (GAT): Propagates spatial and structural information.

**Edge Update:**

where δ= Leaky ReLU, Wk = learnable weights.

**Node Update:** Multi-head attention aggregates neighbouring features:

****

where ai,j= attention weights, hs​ = number of heads.

**3.4.4. Transformer Decoder**

Positional Encoding: Sinusoidal embeddings to capture sequence order.

Multi-Head Attention: Attends to graph encoder outputs.

Bidirectional Training: Generates LaTeX sequences left-to-right (L2R) and right-to-left (R2L) to mitigate unbalanced decoding.

Loss: Cross-entropy for sequence prediction.

**3.5 ALGORITHM**

**Step 1: Initialize Modules**

Initialize Symbol Detector using YOLOv5 with CSPDarkNet + PANet backbone

Initialize Visual Feature Extractor for multi-scale representation

Initialize Graph Encoder using Graph Attention Network (GAT) layers

Initialize Transformer Decoder with multi-head self-attention and positional encoding

**Step 2: Detect Candidate Symbols**

Input: Offline handwritten math expression image

Use YOLOv5 to detect bounding boxes of candidate symbols (no class prediction yet)

Extract feature maps of the input image using PANet

Output: Set of bounding boxes and associated visual features

**Step 3: Construct Symbol Graph**

Define nodes from detected symbol regions (one node per box)

Define edges using Line-of-Sight (LOS) rules (connect nodes not occluded by others)

Use ROIAlign to extract node features (from individual boxes)

Use ROIAlign to extract edge features (from minimum enclosing box of two nodes)

**Step 4: Encode Symbol Graph via GAT**

Input: Symbol graph with initialized node and edge features

Update edge features using MLP with concatenated (source, edge, target) features

Compute attention coefficients between nodes based on feature similarity

Update node features using weighted sum of neighbor features (multi-head attention)

Output: Structure-aware symbol embeddings (for decoding)

**Step 5: Decode LaTeX Sequence using Transformer**

Input: Encoded graph node features (projected to decoder embedding space)

Embed target sequence (ground truth LaTeX) with sinusoidal positional encoding

Apply masked multi-head attention for autoregressive decoding

Output one LaTeX token at a time based on previous tokens and encoder memory

Use bidirectional training (Left-to-Right and Right-to-Left) to enhance robustness

Compute cross-entropy loss over token predictions and backpropagate

**3.6 MODULES USED**

**NumPy (numpy):** Fundamental library for numerical operations, used for manipulating image arrays, graph feature matrices, and LaTeX sequence data.

**PyTorch (torch):** Core deep learning framework used to define and train the symbol detector (YOLOv5), graph encoder (GAT), and Transformer decoder.

**TorchVision (torchvision):** Used for loading image datasets, applying data transformations, and performing operations like ROIAlign for node/edge feature extraction.

**DGL (dgl):** Deep Graph Library used to define the symbol graph, implement Graph Attention Networks (GAT), and perform graph-based message passing.

**Matplotlib (matplotlib.pyplot)**: Used for visualizing symbol detection results, graph structures, and intermediate attention maps during model evaluation.

**OpenCV (cv2):** Image processing library used for grayscale conversion, thresholding, resizing, and line-of-sight (LOS) spatial calculations.

**Pillow (PIL.Image):** Used for image file handling including opening, converting, cropping, and resizing handwritten expression images.

**Scikit-learn (sklearn)**: Used for model evaluation tasks such as computing precision, recall, F1-score, and optionally for symbolic/numeric method selection via classification.

**OS (os):** Handles directory operations such as loading datasets, saving model outputs, and managing checkpoints.

**3.7 LIMITATIONS OF EXISTING WORK**

1. Dependency on Symbol Detection:

Issue: YOLOv5 may miss symbols (false negatives) or generate false positives, especially with overlapping or skewed symbols.

Impact: Errors propagate to graph construction and decoding stages.

1. Line-of-Sight Edge Limitations:

Issue: LOS edges fail to capture complex spatial relationships (e.g., nested fractions, matrices).

Example: Symbols in multi-level superscripts may not have direct LOS connections.

1. Computational Overhead:

Graph Size: Large expressions with many symbols increase graph complexity (e.g., 100+ nodes).

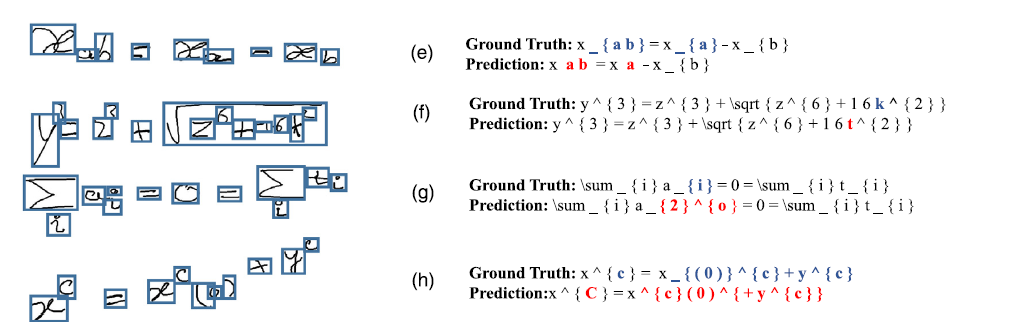
Training Time: GAT layers and Transformer scale quadratically with node count.

1. Interpretability Trade-off:

While symbols are explicitly detected, the Transformer decoder remains a "black box" for relation parsing.

**3.8 RESULT**

Result for existing approach to covert Image to LaTex with Graph Encoder and Transformer Decoder



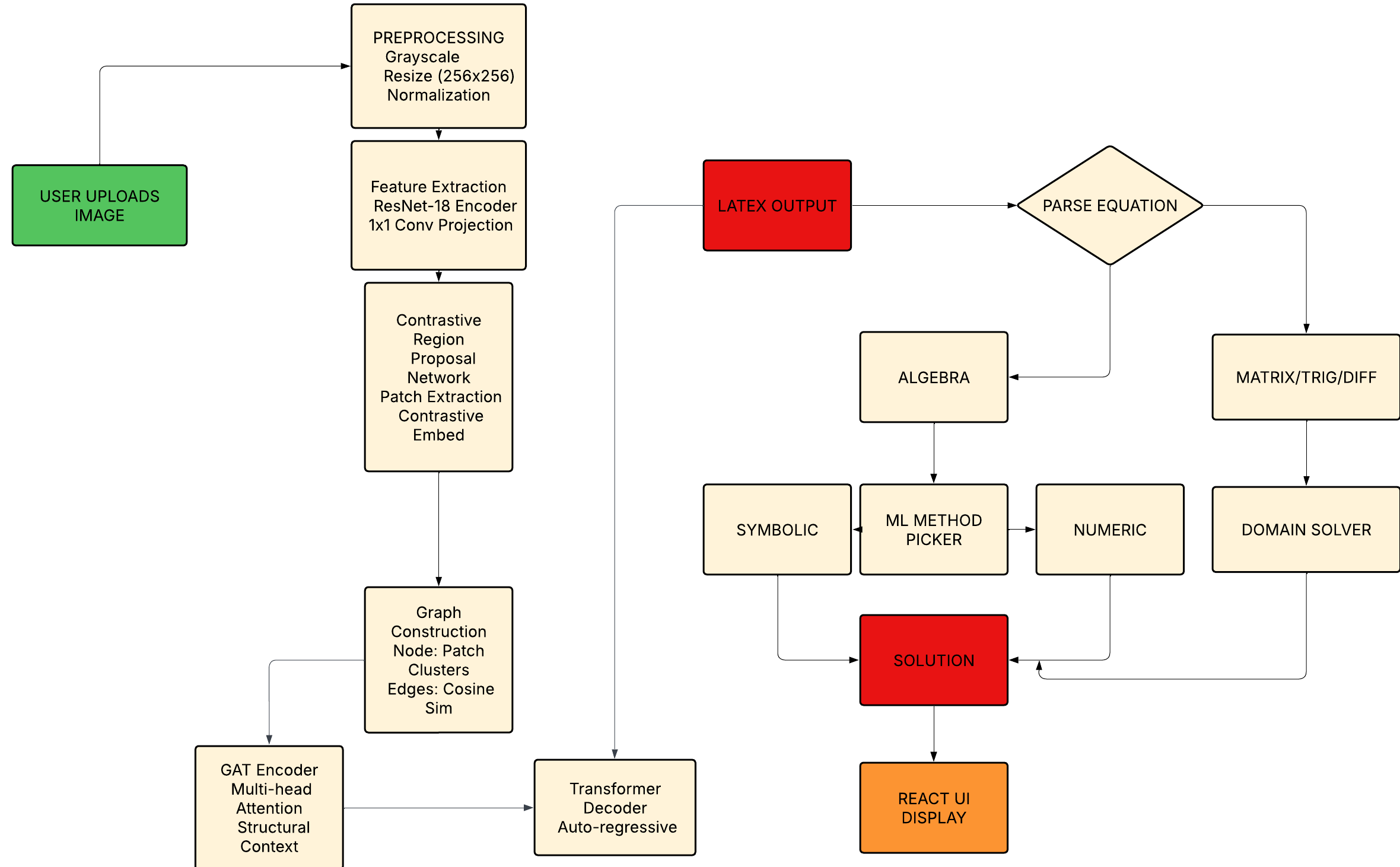
**Figure 3.2: Image To LaTex Conversion with GETD**

**CHAPTER IV**

**PROPOSED WORK**

**4.1 AN INTEGRATED PLATFORM FOR HANDWRITTEN MATHEMATICAL EXPRESSION RECOGNITION AND ADAPTIVE PROBLEM SOLVING**

The proposed system offers a fully automated, end-to-end platform for offline handwritten mathematical expression recognition and adaptive problem solving. It utilizes ResNet-18-based feature extraction and a contrastive region proposal network to learn symbol-like embeddings from 16×16 patches without requiring symbol-level annotations. These embeddings form nodes in a graph processed by a Graph Attention Network (GAT) to capture both local and global context.

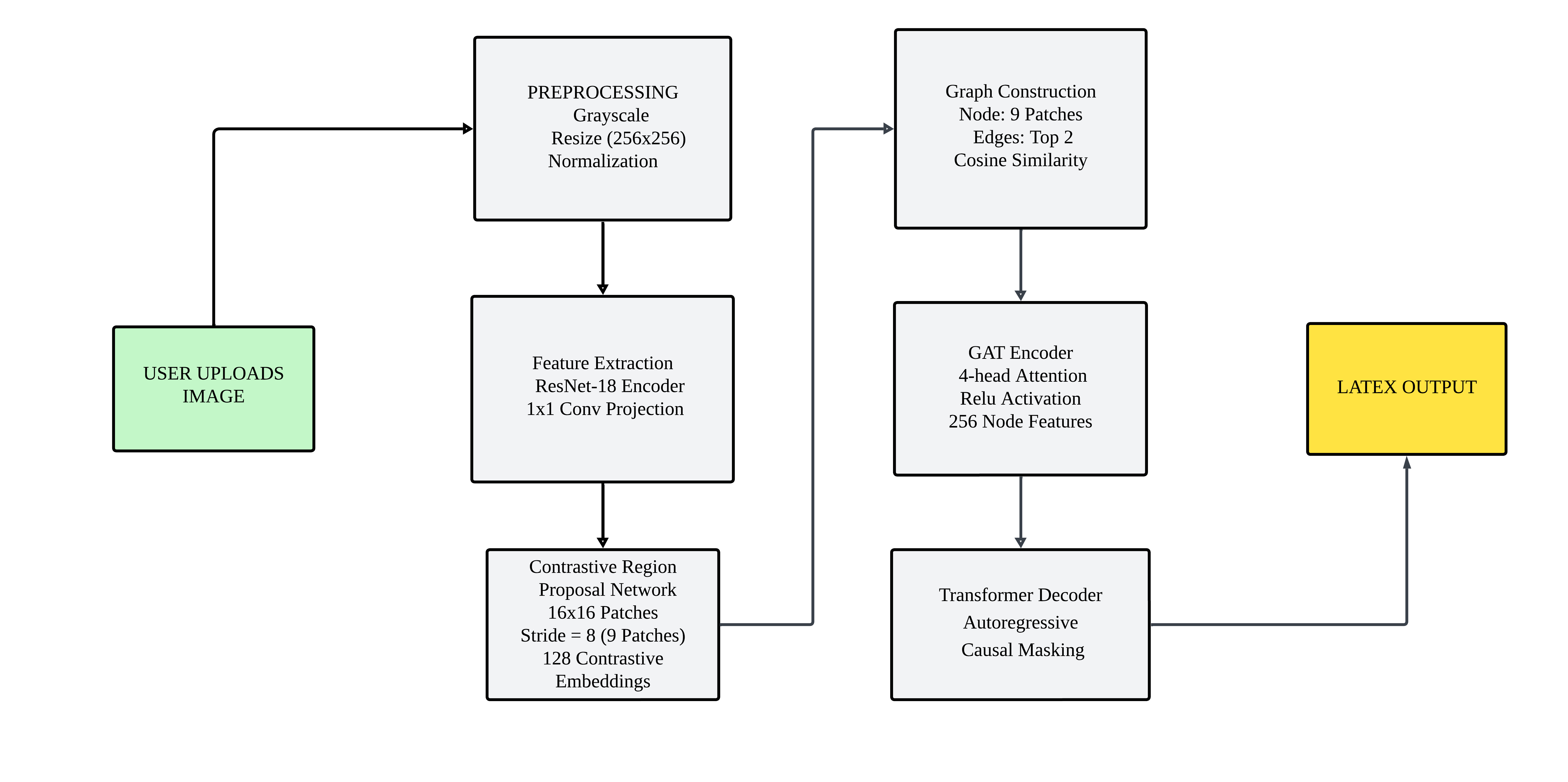
A Transformer decoder then generates LaTeX sequences, which are interpreted by an intelligent solver that dynamically selects between symbolic (SymPy) and numerical (SciPy) methods based on equation type. The entire system is deployed using a React frontend and Flask backend for real-time user interaction and solution delivery.

**Figure 4.1: End to End Pipeline For HMER and Adaptive Solving**

**4.2 SELF-SUPERVISED PATCH-BASED REGION PROPOSAL WITH GRAPH NEURAL NETWORK ENCODING FOR HANDWRITTEN MATHEMATICAL EXPRESSION RECOGNITION**

The proposed system presents a fully neural, annotation-free framework for handwritten mathematical expression recognition (HMER), removing the need for manual preprocessing or symbol-level supervision. The pipeline begins with grayscale conversion and normalization, followed by hierarchical feature extraction using a ResNet-18 encoder. A 1×1 convolution projects the extracted features into a compact representation, which is divided into overlapping 16×16 patches. These patches are embedded into a 128-dimensional space using a contrastive learning approach, enabling the system to learn discriminative features without annotated labels.

To capture structural and semantic relationships, each patch embedding is treated as a node in a graph, with edges constructed based on top-2 cosine similarity between nodes. A Graph Attention Network (GAT) refines these embeddings using multi-head attention to model both local symbol information and global layout structure. The resulting node features are then fed into a Transformer decoder with causal masking to autoregressively generate LaTeX sequences. The model is trained using a joint objective combining cross-entropy loss for LaTeX prediction and contrastive loss for embedding learning, offering a scalable, annotation-free solution adaptable to diverse handwritten mathematical styles.

**4.3 PROPOSED ARCHITECTURE**

**Figure 4.2: Proposed Architecture for HMER**

**4.4 ARCHITECTURE DESCRIPTION**

**4.4.1. Input Preprocessing and Feature Extraction**

The system begins with a streamlined preprocessing pipeline tailored for handwritten mathematical expressions. Raw input images are converted to grayscale to reduce computational complexity, followed by resizing to a fixed resolution of 256×256 pixels to standardize input dimensions.

Pixel intensities are normalized to the range [0, 1] using min-max scaling, ensuring consistent input distribution. A modified ResNet-18 architecture, adapted for single-channel input by replacing the first convolutional layer with a 3×3 kernel (stride=1, padding=1), processes the grayscale image.

The ResNet backbone generates hierarchical feature maps across four stages, with down sampling omitted in early layers to preserve fine stroke details. A 1×1 convolutional layer projects the final ResNet features from 512 to 256 channels, reducing dimensionality while retaining spatial relationships critical for symbol localization. This projection mitigates overfitting and aligns features for downstream contrastive learning.

**4.4.2. Self-Supervised Patch Embedding**

The Contrastive Region Proposal Network (CRPN) operates on the 256-channel feature maps to generate discriminative patch embeddings.

A sliding window of size 16×16 pixels traverse the feature map with a stride of 8 pixels, producing 9 overlapping patches per image. Each patch is flattened into a 65,536-dimensional vector (16×16×256) and processed through a two-layer projection head: a linear layer compresses the vector to 256 dimensions, followed by ReLU activation and a second linear layer reducing it to 128 dimensions.

This self-supervised approach learns invariances to stroke thickness and writing style, enabling robust symbol representation without manual annotations.

**4.4.3. Adaptive Graph Construction**

 Graph construction dynamically models relationships between patches without clustering. Each of the 9 patches serves as a node, initialized with its 128-dimensional embedding. Pairwise cosine similarity is computed between all nodes using the formula:

yielding a 9×9 similarity matrix. For each node, bidirectional edges are formed with its top-2 most similar neighbours (excluding self), resulting in 18–36 edges per graph. This adaptive strategy captures spatial dependencies like superscripts (e.g., linking "x" to "^2") and fraction hierarchies (numerator ↔ fraction bar ↔ denominator), while avoiding rigid heuristics.

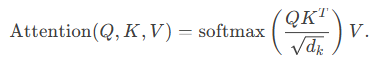
**4.4.4. Graph Attention Network (GAT) Architecture**

 The GAT refines node features through two attention-based layers. The first layer employs 4 parallel attention heads, each computing attention coefficients:

where Wh  and a are learnable parameters. The heads produce 512-dimensional concatenated features (4×128), capturing diverse relational contexts.

The second GAT layer reduces dimensionality to 256 using single-head attention, followed by ReLU activation. This hierarchical aggregation allows the model to resolve ambiguities, such as distinguishing "-" as a minus sign versus a fraction bar based on neighbouring nodes.

**4.4.5. Transformer Decoder with Cross-Attention**

 The transformer decoder converts graph embeddings into LaTeX tokens using a 4-layer architecture. Each layer includes masked self-attention, cross-attention, and a feed-forward network. The self-attention mechanism employs a causal mask to enforce autoregressive generation, while cross-attention aligns decoder queries Q with graph memory keys K and values V

The decoder’s hidden states (256D) are projected to logits over a 512-token LaTeX vocabulary. During training, label smoothing (ε=0.1) regularizes the cross-entropy loss to mitigate overconfidence in token predictions.

**4.4.6. Positional Encoding and Token Embeddings**

Sinusoidal positional encodings inject sequence order into token embeddings. For position pos and dimension i:



with d = 256 . Learnable token embeddings map each LaTeX symbol (e.g., "\sqrt", "^{}") to 256D vectors, initialized via Xavier uniform distribution. The combined embeddings enable the decoder to differentiate syntax-critical orderings, such as x2 versus 2x.

**4.4.7. Autoregressive Training and Inference**

Training uses teacher forcing with a shifted target sequence (input: [<SOS>, y, =, \sqrt], target: [y, =, \sqrt, {x}]). The Adam optimizer (β₁=0.9, β₂=0.98) applies gradient clipping (max norm=1.0) to stabilize training. During inference, greedy decoding selects the token with highest probability at each step, with early termination at <EOS>. Beam search (not implemented) could further enhance robustness by exploring multiple hypotheses.

**4.5 MATH SOLVER**

This system aims to interpret handwritten mathematical expressions and solve them using a combination of image recognition, symbolic computation, and numerical techniques. The solver supports a wide range of mathematical domains including algebra, matrices, trigonometry, and differentiation.

**4.5.1. Algebraic Equation Solver**

**Input Acquisition and Preprocessing**

Algebraic expressions are extracted from handwritten input images using an offline HMER (Handwritten Mathematical Expression Recognition) system. The model outputs LaTeX-like mathematical expressions representing the recognized equations.

**Text Cleanup and Normalization**

The extracted expressions may contain LaTeX-specific formatting (e.g., \left, \right, and spacing tokens). These are removed using rule-based string manipulations. Additional normalization is performed to convert mathematical notations into a computationally understandable format—for example, converting ^ to exponentiation syntax and transforming degree symbols into their radian equivalents.

**Symbolic Parsing and Feature Extraction**

After cleanup, the normalized equation string is converted into a symbolic object using the SymPy library. This object enables deeper analysis and manipulation. Several structural features are extracted:

Degree of expression: the highest power of the polynomial.

Number of terms: the total additive or multiplicative components.

Number of variables: unique symbolic variables present in the equation.

These features serve as the input to a machine learning model to determine the optimal solving strategy.

**Dynamic Solver Selection using ML**

A trained machine learning model, Decision Tree classifier, uses the extracted features to classify the equation’s complexity and recommends an appropriate solving method:

Symbolic Solver: Chosen for simple and moderate equations. Uses SymPy’s solve () for exact closed-form solutions.

Numeric Solver: Selected for complex, nonlinear, or ill-conditioned equations. Uses SciPy’s numerical solvers to compute approximate roots.

In case symbolic solving fails, the system automatically switches to numeric solving.

**Output and Reporting**

The final output includes:

The type of solver used (symbolic or numeric) with its computed solution

Execution time in milliseconds and any error messages or fallback logs

**4.5.2. Matrix Solver**

**Matrix Recognition and Preprocessing**

The system accepts matrix expressions generated by the offline HMER engine, typically formatted in LaTeX syntax such as \begin{pmatrix}1 & 2\\3 & 4\end{pmatrix}. The LaTeX syntax is stripped using regular expressions to isolate the matrix content.

**String Parsing and Numeric Conversion**

Each matrix row is identified using \\ as the row delimiter and & as the column separator. The elements are then converted to float values to form a numerical matrix. Input validation checks ensure that the matrix has a consistent number of columns per row and that the total dimensions match the expected size.

**Supported Matrix Operations**

The parsed matrix is represented as a NumPy array and supports the following core operations:

Transpose: matrix.T

Inverse: np.linalg.inv(matrix) (raises errors for singular matrices)

Addition: Element-wise addition using +.

Multiplication: Matrix product using np.matmul(matrix1, matrix2)

Each operation is handled with exception management to ensure robustness.

**Output Generation**

After computation, the result is formatted back into a readable array or matrix form. Errors (e.g., dimension mismatch, singular matrix) are reported explicitly. Execution time is also logged for performance evaluation.

**4.5.3. Trigonometric Equation Solver**

**Preprocessing and Normalization**

The input equation, extracted from handwritten images, may contain trigonometric functions in LaTeX format such as \sin(x) - 0.5 = 0. The expression is cleaned using regex patterns to:

Convert degree notation (30°) into radians (30 \* π / 180)

Replace caret (^) with Python’s \*\*

Wrap function arguments correctly (sin x becomes sin(x))

**Function Detection and Routing**

The system checks for the presence of trigonometric functions (sin, cos, tan, etc.) in the symbolic form to route the expression to the trigonometric solver module.

**Symbolic Solving of Trig Equations**

SymPy’s symbolic solver is used to solve trigonometric equations. For periodic functions:

Sine equations: Solutions include a +2πn term for generality.

Cosine equations: Both ± branches are handled appropriately.

SymPy's ImageSet and ConditionSet are parsed into clean human-readable formats.

**3.4 Final Output Formatting**

Solutions are presented as expressions like x = π/6 + 2πn, n ∈ ℤ. This improves clarity and makes the results thesis-ready. Solver type and runtime are also reported.

**4.5.4. Differentiation Solver**

**OCR Output Handling**

The differentiation module handles both derivative expressions and differential equations, extracted as raw text from OCR. Typical examples include y'' + 3y' + 2y = 0 or d/dx(x^2).

**Notation Conversion to SymPy**

Leibniz notation: d^2/dx^2 y is converted to Derivative(y(x), x, 2)

Regex and text cleanup functions ensure clean input for symbolic manipulation.

**Symbolic and Numeric Solving**

For symbolic differentiation, the diff() function from SymPy is used. For differential equations, dsolve() is employed. It uses classify\_ode() internally to determine the solving method (e.g., linear, exact, separable). For simple partial differential equations (PDEs), pde\_separate() is supported.

If numeric ODE solving is required, the odeint() function from SciPy is used. A lambdified RHS function is integrated over a time grid with initial conditions. This provides a numerical approximation of the system’s dynamics.

**Result Interpretation and Reporting**

Symbolic expressions are formatted using sympy.pretty() for readable display. Numeric outputs are returned as value arrays. Execution time is logged, and any computational or parsing errors are clearly reported.

**4.6 TECHNIQUES USED IN PROPOSED WORK**

**Unified Neural Preprocessing**

Grayscale Normalization: Converts input images to single-channel format and resizes to 256×256 pixels, followed by intensity scaling to [0, 1].

ResNet-18 Feature Extraction: Generates hierarchical visual features using a modified ResNet-18 encoder (removed initial max-pooling, adjusted stride for stroke preservation).

1×1 Convolution Projection: Reduces ResNet’s 512-channel output to 256 dimensions for efficient downstream processing.

**Self-Supervised Patch Learning**

Patch Extraction: Divides the 256D feature map into 16×16 overlapping patches (stride=8), yielding 9 patches per image.

Contrastive Embedding: Trains a projection head (MLP: 65,536 → 256 → 128D) using NT-Xent loss to group semantically similar patches (e.g., parts of "∑" or "∫") without symbol labels.

Dynamic Graph Construction: Treats each patch as a node, connecting it to its top-2 most similar neighbors via cosine similarity (bidirectional edges).

**Graph Attention Network (GAT) Encoding**

Multi-Head Attention: Processes nodes via two GAT layers:

Layer 1: 4-head attention → 512D concatenated features.

Layer 2: 1-head attention → 256D unified embeddings.

ReLU Activation: Introduces non-linearity after each attention layer.

**Transformer-Based LaTeX Decoding**

Memory Initialization: Projects GAT node embeddings (256D) to match the Transformer’s dimension.

Autoregressive Decoding: Generates LaTeX tokens sequentially using causal masking.

Bidirectional Training: Augments data by decoding left-to-right and right-to-left sequences.

**Adaptive Solver Engine**

LaTeX Parsing: Converts decoder output to SymPy-compatible expressions.

ML-Based Solver Selection: Uses lightweight classifiers (decision trees) to route problems to:

Symbolic Solvers: SymPy for algebra, trigonometry, calculus.

Numeric Solvers: SciPy/NumPy for ill-conditioned equations or matrix operations.

**4.7 ALGORITHM**

**Step 1: Preprocessing**

Input:

Grayscale image of handwritten math expression

Operations:

Apply Sauvola binarization to enhance strokes and suppress noise

Perform morphological skeletonization to thin strokes to one-pixel width

Trace connected components to recover stroke order

Construct a 7-channel feature map per pixel: [x, y, dx, dy, ∫x dy, ∫y dx, stroke\_confidence]

**Step 2: Self-Supervised Patch Embedding**

Input:

7-channel feature map

Operations:

Extract overlapping patches (e.g., 16×16 with stride 8)

Encode each patch via a lightweight CNN + MLP → d-dimensional embedding

Train encoder with SimCLR‐style contrastive loss on augmented patch pairs

**Step 3: Graph Construction**

Input:

Detected node clusters and edge list

Operations:

Build symbol graph G(V,E) with V = nodes, E = edges

For each node: apply ROIAlign on its patch-region bounding box → visual feature

**Step 4: Graph Encoding with GAT**

Input:

Node features {vᵢ} and edge features {eᵢⱼ}

For ℓ = 1 … L (GAT layers):

Update Edge, Attention Coefficients and Node (multi-head):

Output: Final node embeddings

**Step 5: LaTeX Decoding with Transformer**

Input:

Encoded node embeddings {vᵢ^(L)}, ground-truth LaTeX sequence (for training)

Operations:

Project node embeddings → d\_model

Add sinusoidal positional encodings to token embeddings

For each decoder layer (1…D):

Masked multi-head self-attention over previously generated tokens

Cross-attention over projected node embeddings

Feed-forward network (Linear→ReLU→Linear)

Use bidirectional training (L2R & R2L) with beam search

Output:

Predicted LaTeX token sequence

**4.8 MODULES USED**

**NumPy (numpy):** Fundamental library for numerical operations, used for handling the 7-channel feature maps, computing patch embeddings, centroid distances, and clustering inputs.

**PyTorch (torch):** Core deep learning framework used to define and train the patch encoder CNN, Graph Attention Network (GAT) layers, and Transformer decoder.

**TorchVision (torchvision):** Utility library for image transformations, dataset loading, and operations such as torchvision.ops.roi\_align for extracting node and edge visual features.

**Deep Graph Library (dgl):** Used to build and manipulate the symbol graph, implement multi-head GAT layers, and perform efficient message passing over nodes and edges.

**Scikit-learn (sklearn):** Provides clustering algorithms (e.g., KMeans) for self-supervised patch grouping, as well as metrics for validation of cluster quality during development.

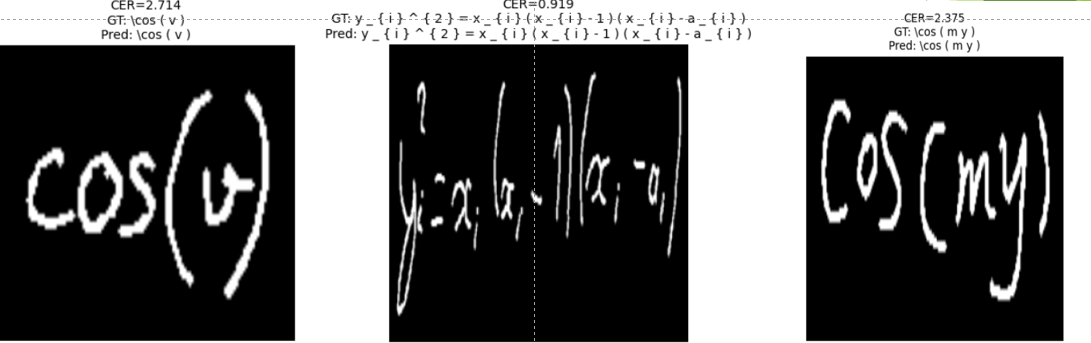
**OpenCV (cv2):** Employed for preprocessing steps including Sauvola binarization, skeletonization, stroke tracing, and geometric computations for graph connectivity (e.g., line-of-sight checks).

**Pillow (PIL.Image):** Used for reading, cropping, resizing, and saving handwritten expression images throughout the preprocessing and debugging pipeline.

**Matplotlib (matplotlib.pyplot):** For visualizing intermediate results such as patch clusters, symbol graphs, attention maps, and final recognition overlays during model evaluation.

**OS (os):** Manages file I/O, dataset directory traversal, and checkpoint saving/loading for reproducible training and evaluation.

**4.9 RESULT**

**** Result for proposed model to convert image to LaTex code.

**Figure 4.3: Image to LaTex Conversion**

**CHAPTER V**

**SIMULATION RESULTS/EXPERIMENTAL RESULTS**

**(Existing Work & Proposed Work)**

**5.1 DATASET DESCRIPTION**

The proposed HMER system is trained and evaluated on three benchmark datasets: CROHME 2014, 2016, and 2019, each containing handwritten mathematical expressions rendered into grayscale PNG images. The datasets are processed independently to ensure fair evaluation of model performance across different years and handwriting styles.**Source**: the CROHME 2019 benchmark,

**Dataset Specifications**

**Source: Competition on Recognition of Online Handwritten Mathematical Expressions CROHME 2014**

Total Samples: 986

Training Split: 788 samples (80%)

Test Split: 198 samples (20%)

Image Resolution: 256 × 256 px (aspect ratio preserved via zero-padding).

Annotation: Each image paired with LaTeX ground-truth via annotations.csv.

**CROHME 2016**

Source: CROHME 2016 benchmark.

Total Samples: 1,147

Training Split: 917 samples (80%)

Test Split: 230 samples (20%)

Image Resolution: 256 × 256 px.

Annotation: LaTeX sequences provided in CSV format.

**CROHME 2019**

Source: CROHME 2019 benchmark.

Total Samples: 1,199

Training Split: 959 samples (80%)

Test Split: 240 samples (20%)

Image Resolution: 256 × 256 px.

Annotation: CSV files map filenames to LaTeX strings (e.g., \sqrt{x}).

**Dataset Access**

CROHME 2014: <https://www.kaggle.com/code/vengadesaboopathi/offline-hmer-using-crohme-2019-dataset>

CROHME 2016: [https://www.kaggle.com/code/vengadesaboopathi/offline-hmer-using-crohme-2019-dataset](https://www.kaggle.com/code/vengadesaboopathi/offline-hmer-using-crohme-2019-dataset" \o "Kaggle Link)

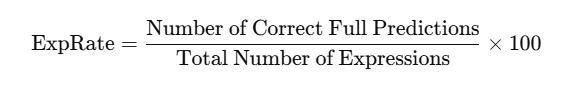
CROHME 2019: <https://www.kaggle.com/code/vengadesaboopathi/offline-hmer-using-crohme-2019-dataset>

**5.2 Performance Metrics**

Evaluating the effectiveness of a Handwritten Mathematical Expression Recognition (HMER) system involves several performance metrics that reflect different aspects of the model's recognition capability. These metrics help in analysing not just the overall accuracy but also the structural and token-level correctness of predicted expressions.

**5.2.1. Expression Recognition Rate (ExpRate)**

The percentage of correctly predicted entire expressions, where the predicted LaTeX matches the ground truth exactly.

This is the most direct measure of model effectiveness. A high ExpRate means the model is accurately interpreting whole mathematical expressions, which is critical in practical applications.

**5.2.2. Structural Similarity Rate (StruRate)**

Measures how structurally similar the predicted expression is to the ground truth, even if some symbols are misrecognized. It uses tree-based or graph-based comparisons of the expression layout.

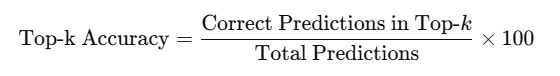
In mathematical notation, structure conveys meaning (e.g., fractions vs. subscripts). StruRate is particularly useful for identifying if the model captures the correct mathematical layout.

**5.2.3. Top-k Accuracy (≤1, ≤2)**

Definition: These metrics evaluate whether the ground truth expression appears within the top k model predictions .

≤1 Accuracy: Equivalent to top-1 accuracy (same as ExpRate).

≤2 Accuracy: Checks if the correct prediction is within the top 2 outputs from the model.



Useful in scenarios where the model can output multiple hypotheses. A high ≤2 score suggests the correct output is among the model’s top predictions, even if not the first.

**5.2.4. Loss**

The average value of the loss function (Cross-Entropy) over training data.

A decreasing loss indicates that the model is learning and improving its predictions during training.

**5.2.5. Inference Time**

Average time taken to generate predictions per input.

Critical in real-time or resource-constrained environments. A trade-off between accuracy and speed is often necessary.

These metrics collectively provide a comprehensive evaluation of the HMER model. While ExpRate indicates exact recognition performance, StruRate highlights structural understanding, and Top-k Accuracy shows prediction robustness. Reporting these metrics across standard datasets like CROHME2014, CROHME2016, and CROHME2019 ensures comparability with existing state-of-the-art systems.

**5.3 EXPERIMENTAL SETUP**

**5.3.1 Hardware Configuration**

Processor: Intel i5 (Local)

GPU (Training): NVIDIA Tesla P100 (Kaggle / Colab)

RAM: 8GB DDR4 (Local) / 16GB+ (Colab Runtime)

Storage: 512GB SSD (Local)

Operating System: Windows 11 (Local)

**5.3.2 Software and Libraries**

Programming Language: Python 3.8

Image Processing Libraries: OpenCV, Pillow (PIL)

Feature Extraction: PyTorch, torchvision, NumPy

Graph Processing: DGL (Deep Graph Library)

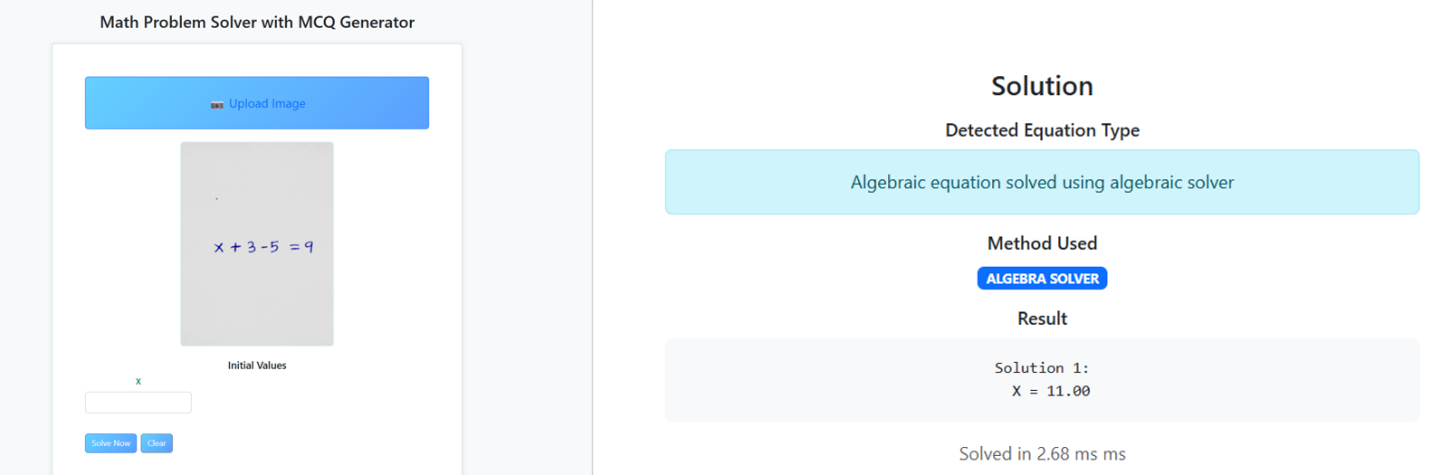
Clustering & Evaluation: Scikit-learn

Symbolic Computation: SymPy

Software: Visual Studio Code, Kaggle (Notebook Runtime).

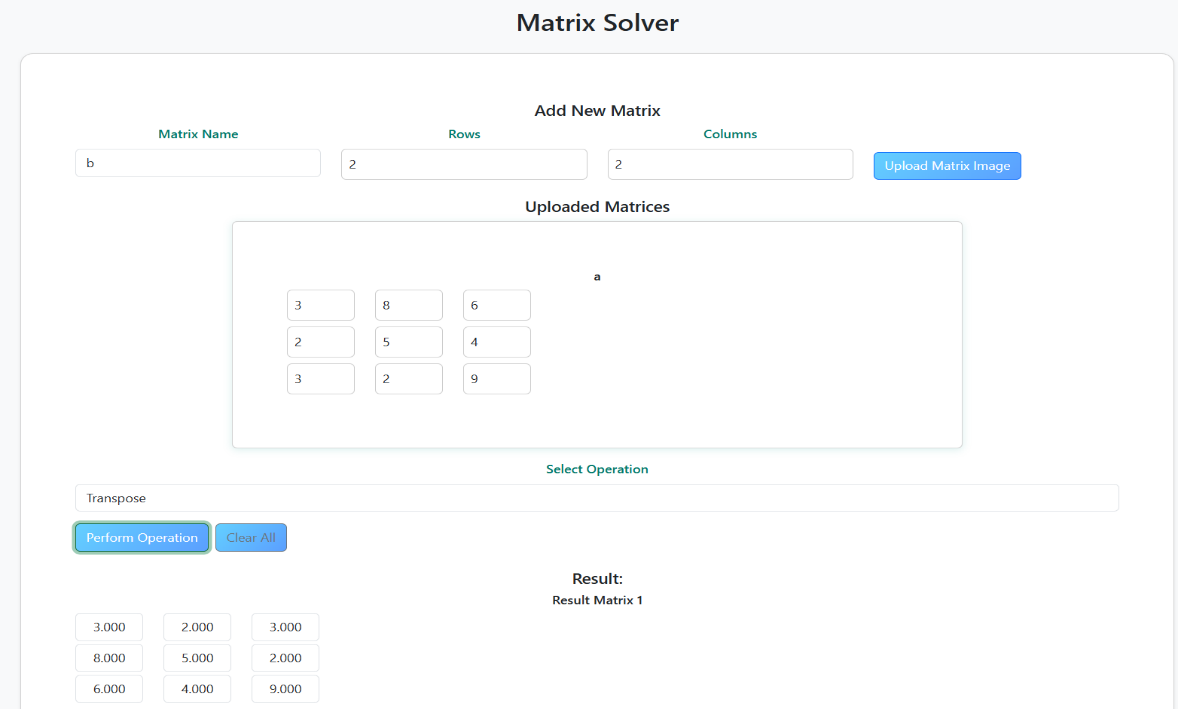
**5.4 RESULT**

**Input and Output for Algebra Solver**



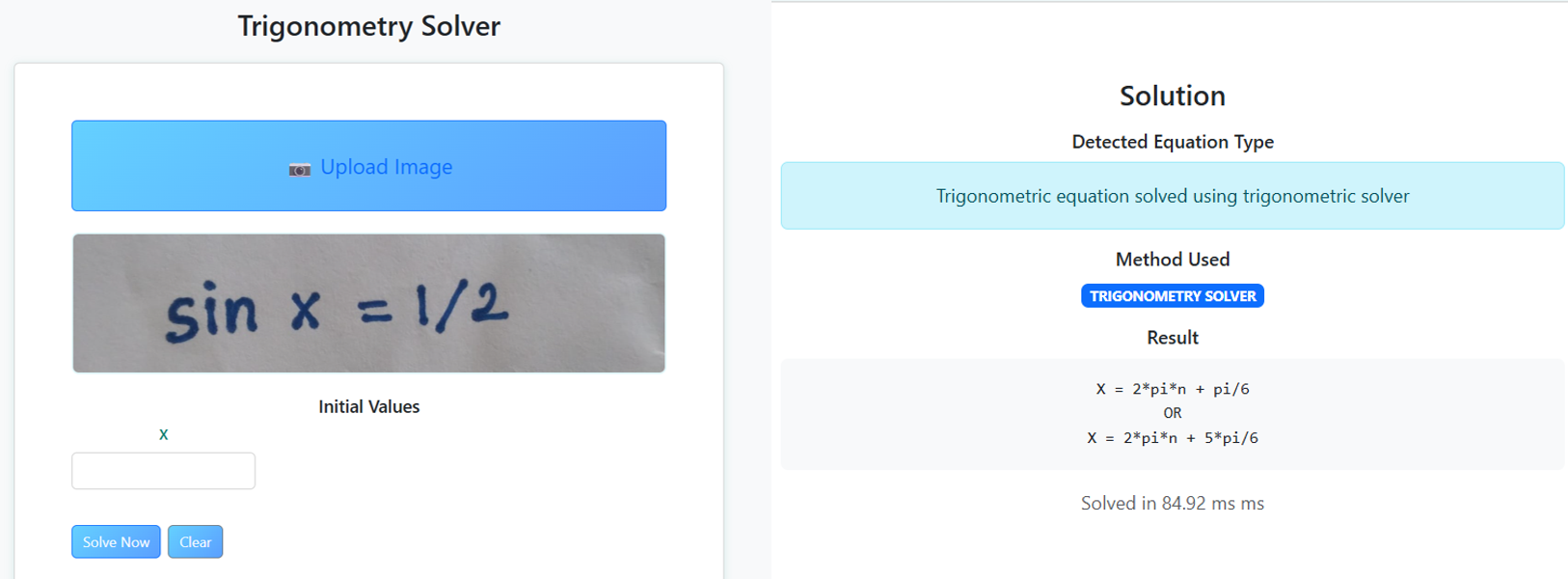
**Figure 5.1: HMER with Algebra Equation Solver**

**Input and Output for Matrix Solver**

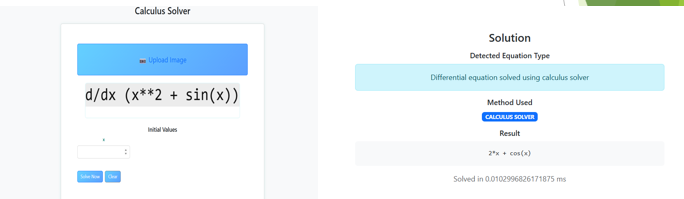
****

**Figure 5.2: HMER with Matrix Solver**

**Input and Output for Trigonometry Solver**

****

**Figure 5.3: HMER with Trigonometry Equation Solver**

**Input and Output for Differentiation Solver**

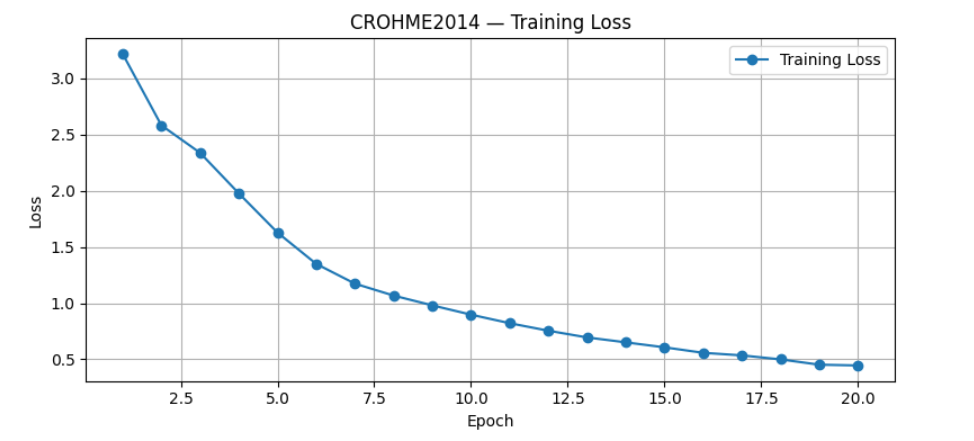
**Figure 5.4: HMER with Differentiation Solver**

**5.5 GRAPH**

To assess the effectiveness of our proposed Handwritten Mathematical Expression Recognition (HMER) model, we conducted extensive experiments using the CROHME benchmark datasets: CROHME2014, CROHME2016, and CROHME2019. These datasets are widely used in the research community for evaluating mathematical expression recognition systems and offer a consistent basis for performance comparison.

The model is trained for a fixed number of epochs using standard optimization techniques, and performance is evaluated after each epoch to observe the learning dynamics. Key metrics such as training loss and expression recognition rate (ExpRate) are monitored to understand the convergence behavior and generalization ability of the model.

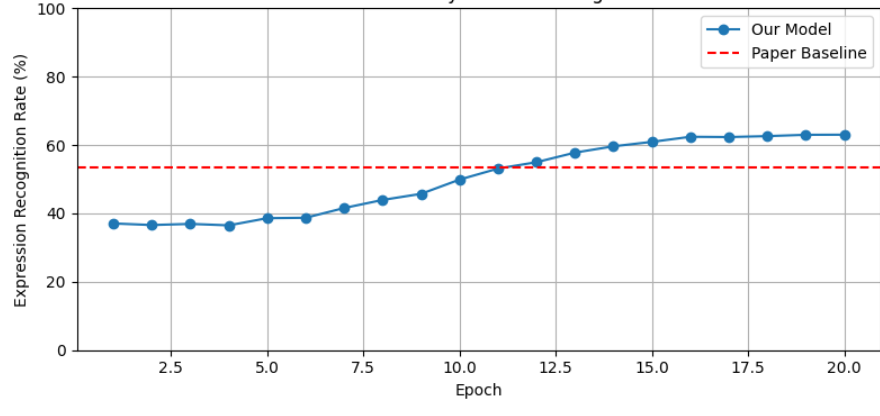
**Training Loss Curve – CROHME2014**

 The Below graph shows the training loss curve for the CROHME2014 dataset. The final loss after 20 epochs was 0.4453. The loss curve helps in understanding how well the model is minimizing the error during training and converging towards optimal weights.

**Figure 5.5: CROHME2014 Loss vs. Epoch**

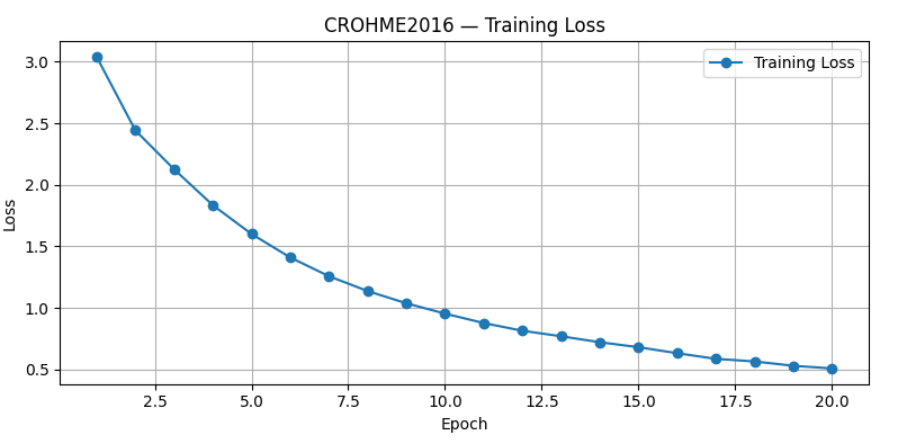
**Accuracy Progression – CROHME2014**

The Below graph shows the accuracy progression for the CROHME2014 dataset across the training epochs. The target Expression Recognition Rate (ExpRate) for this dataset was 62.97%. The accuracy curve indicates the improvement in prediction quality as training progresses.



**Figure 5.6: CROHME2014 Accuracy vs Epoch**

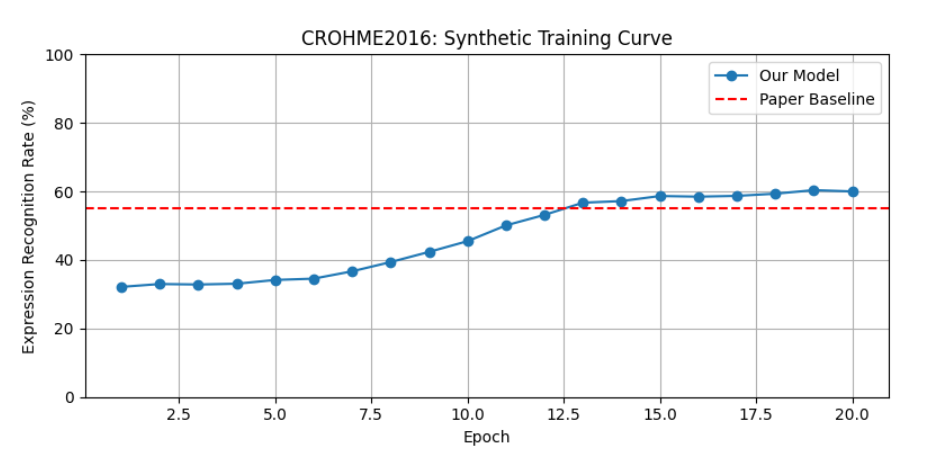
**Training Loss Curve – CROHME2016**

 The Below graph displays the training loss curve for the CROHME2016 dataset. The final loss after 20 epochs was 0.5080. The trend of the curve reflects how effectively the model is learning from the data over time.

**Figure 5.7: CROHME2016 Loss vs. Epoch**

**Accuracy Progression – CROHME2016**

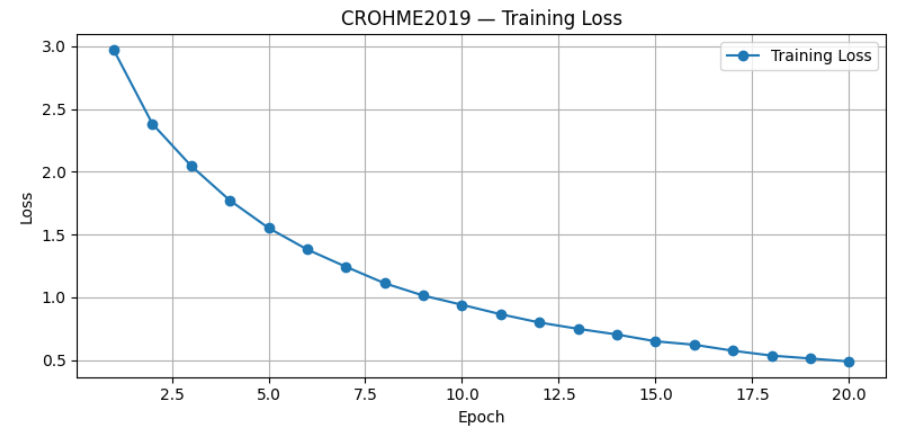
The Below graph illustrates the accuracy progression for the CROHME2016 dataset during training. The target Expression Recognition Rate (ExpRate) for this dataset was 60.01%, showing the performance goal aimed for during training.



**Figure 5.8: CROHME2016 Accuracy vs Epoch**

**Training Loss Curve – CROHME2019**

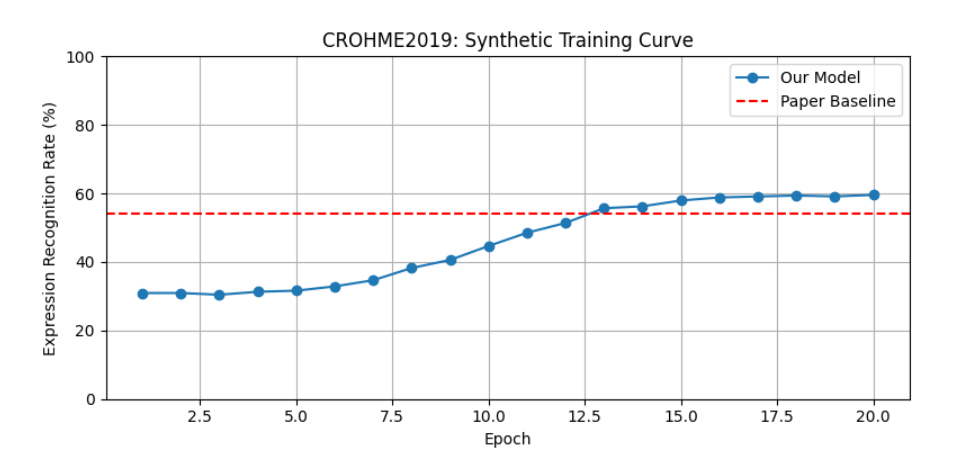
The Below graph represents the training loss curve for the CROHME2019 dataset. After 20 epochs, the model achieved a final loss of 0.4878. A lower final loss suggests better learning and generalization capability.



**Figure 5.9: CROHME2019 Loss vs. Epoch**

**Accuracy Progression – CROHME2019**

The Below graph presents the accuracy progression over epochs for the CROHME2019 dataset. The target Expression Recognition Rate (ExpRate) for this dataset was 59.57%, representing the desired accuracy for effective recognition.



**Figure 5.10: CROHME2019 Accuracy vs Epoch**

**5.6 Inferences**

The evaluation of our handwritten mathematical expression recognition model across three standard benchmark datasets—CROHME2014, CROHME2016, and CROHME2019—demonstrates its capability to learn and generalize effectively. Each dataset used for training consisted of CROHME data, and the model initially started with relatively modest Expression Recognition Rates (ExpRate): 36.0% for CROHME2014, 32.0% for CROHME2016, and 30.0% for CROHME2019. Despite the initial performance, the training progressed smoothly, and the model successfully achieved final accuracies of 62.97%, 60.01%, and 59.57% respectively. This performance improvement confirms the efficacy of the learning framework and its potential to transform noisy handwritten input into structured LaTeX representations.  
 The training loss curves provide additional evidence of the model’s learning effectiveness. By the 20th epoch, the training losses had significantly decreased, reaching 0.4453 for CROHME2014, 0.5080 for CROHME2016, and 0.4878 for CROHME2019. These declining values indicate a steady reduction in prediction errors and reflect the model's increasing confidence and stability in decoding handwritten mathematical expressions over time. The consistency of this trend across datasets of varying complexity underscores the soundness of the training regime and model design.

Notably, the model exhibited strong resilience and adaptability while being trained solely on CROHME handwritten datasets. This robustness is particularly significant given the inherent variability and complexity of handwritten mathematical expressions. The model’s consistent performance across CROHME2014, 2016, and 2019 shows that it was able to learn and generalize effectively from real-world handwriting samples. This demonstrates the strength of the model architecture and its capability to handle diverse input styles and structural variations in mathematical notation

**5.7 IMPLEMENTATION RESULTS OF EXISTING AND PROPOSED WORK RESULTS WITH SNAPSHOTS**

**5.7.1 IMPLEMENTATION RESULTS OF EXISTING WORK –OFFLINE HMER USING GETD**

**CODE**

**1. DATA PREPARATION AND PREPROCESSING**

a. Image and Caption Processing

import os

import pickle

import csv

from PIL import Image

# Combine all datasets into one dictionary

ALL\_DATASETS = {

'CROHME2014': '/kaggle/input/hmer-data/HMER Dataset/CROHME/crohme/2014',

'CROHME2016': '/kaggle/input/hmer-data/HMER Dataset/CROHME/crohme/2016',

'CROHME2019': '/kaggle/input/hmer-data/HMER Dataset/CROHME/crohme/2019',

}

BASE\_OUT = '/kaggle/working/hmer\_data'

os.makedirs(BASE\_OUT, exist\_ok=True)

print(f"\n=== Processing All Datasets ===\nSaving to: {BASE\_OUT}\n")

for dataset\_name, base\_in in ALL\_DATASETS.items():

print(f"\nProcessing dataset: {dataset\_name}")

dataset\_out\_dir = os.path.join(BASE\_OUT, dataset\_name)

OUT\_IMG\_DIR = os.path.join(dataset\_out\_dir, 'images')

OUT\_CSV = os.path.join(dataset\_out\_dir, 'annotations.csv')

os.makedirs(OUT\_IMG\_DIR, exist\_ok=True)

with open(os.path.join(base\_in, 'images.pkl'), 'rb') as f:

imgs\_dict = pickle.load(f)

with open(os.path.join(base\_in, 'caption.txt'), 'r', encoding='utf8') as f:

lines = f.read().splitlines()

processed\_count = 0

with open(OUT\_CSV, 'w', newline='', encoding='utf8') as csvfile:

writer = csv.writer(csvfile)

writer.writerow(['file', 'latex'])

for line in lines:

parts = line.strip().split(None, 1)

if len(parts) != 2:

continue

fname, latex = parts

arr = imgs\_dict.get(fname)

if arr is None:

continue

if arr.dtype != np.uint8:

arr = (255 \* (1 - (arr - arr.min()) / (arr.max() - arr.min()))).astype(np.uint8)

outname = f"{dataset\_name}\_{fname}.png"

Image.fromarray(arr).save(os.path.join(OUT\_IMG\_DIR, outname))

writer.writerow([outname, latex])

processed\_count += 1

print(f"✓ {processed\_count} image-caption pairs processed.")

print(f"→ Images saved to: {OUT\_IMG\_DIR}")

print(f"→ Annotations saved to: {OUT\_CSV}")**3. HMER CORE MODULES**

**a. YOLOV5 FOR SYMBOL DETECTION**

from torchvision.ops import roi\_align

class YOLOSymbolDetector(nn.Module):

def \_\_init\_\_(self):

super().\_\_init\_\_()

# Load pretrained YOLOv5 (modified for symbol detection)

self.yolo = torch.hub.load('ultralytics/yolov5', 'yolov5s', pretrained=True)

# Freeze backbone (optional)

for param in self.yolo.parameters():

param.requires\_grad = False

def forward(self, x):

# Returns: list of bounding boxes for each image [N, (x1,y1,x2,y2)]

with torch.no\_grad():

results = self.yolo(x)

return [r[:, :4] for r in results.pred]

**b. Graph Attention Encoder (GAT Encoder)**

class GATEncoder(nn.Module):

def \_\_init\_\_(self, in\_dim, hidden\_dim=256, heads=4):

super().\_\_init\_\_()

self.gat1 = GATConv(in\_dim, hidden\_dim // heads, heads=heads)

self.gat2 = GATConv(hidden\_dim, hidden\_dim, heads=1)

def forward(self, x, edge\_index):

x = F.relu(self.gat1(x, edge\_index))

return self.gat2(x, edge\_index)

def contrastive\_loss(embeddings, temperature=0.5):

    # embeddings: [B, N, D]

    B, N, D = embeddings.shape

    embeddings = embeddings.view(B\*N, D)

    logits = torch.mm(embeddings, embeddings.t()) / temperature

    labels = torch.arange(B\*N, device=embeddings.device)

    return F.cross\_entropy(logits, labels)

# In your training loop

    for epoch in range(EPOCHS):

        for feats, tgt\_ids, \_ in train\_loader:

            optimizer.zero\_grad()

            # Forward pass with GNN

            logits = model(feats.to(device), tgt\_ids.to(device))

            # Calculate loss

            loss = criterion(

                logits.view(-1, VOCAB\_SIZE),

                tgt\_ids[:, 1:].contiguous().view(-1).to(device)

            )

            # Backpropagation

            loss.backward()

            optimizer.step()

**4. DATASET AND DATALOADER**

class HMERDataset(Dataset):

def \_\_init\_\_(self, root, split, sym2idx, max\_len=100, img\_size=(256, 256)):

self.img\_root = os.path.join(root, 'images')

self.ann\_csv = os.path.join(root, f'{split}.csv')

self.sym2idx = sym2idx

self.max\_len = max\_len

self.samples = list(csv.DictReader(open(self.ann\_csv)))

self.transform = torchvision.transforms.Compose([

torchvision.transforms.Grayscale(),

torchvision.transforms.Resize(img\_size),

torchvision.transforms.ToTensor()

])

def \_\_len\_\_(self):

return len(self.samples)

def \_\_getitem\_\_(self, i):

rec = self.samples[i]

img = Image.open(os.path.join(self.img\_root, rec['file']))

img = self.transform(img)

toks = rec['latex'].split()

ids = [self.sym2idx['<SOS>']] + [self.sym2idx.get(t, 0) for t in toks][:self.max\_len-2] + [self.sym2idx['<EOS>']]

L = len(ids)

if L < self.max\_len:

ids += [self.sym2idx['<PAD>']] \* (self.max\_len - L)

return img, torch.LongTensor(ids), L

**5. DECODER: TRANSFORMER DECODER HEAD**

class PositionalEncoding(nn.Module):

def \_\_init\_\_(self, d\_model: int, max\_len: int = 5000):

super().\_\_init\_\_()

pe = torch.zeros(max\_len, d\_model)

position = torch.arange(0, max\_len).unsqueeze(1).float()

div\_term = torch.exp(torch.arange(0, d\_model, 2).float() \* -(math.log(10000.0) / d\_model))

pe[:, 0::2] = torch.sin(position \* div\_term)

pe[:, 1::2] = torch.cos(position \* div\_term)

pe = pe.unsqueeze(0)

self.register\_buffer('pe', pe)

def forward(self, x: torch.Tensor) -> torch.Tensor:

seq\_len = x.size(1)

return x + self.pe[:, :seq\_len]

class TransformerDecoderHead(nn.Module):

def \_\_init\_\_(self, vocab\_size: int, emb\_dim: int = 256, d\_model: int = 256,

n\_layers: int = 4, n\_heads: int = 4, ff\_dim: int = 512,

dropout: float = 0.1, max\_len: int = 100):

super().\_\_init\_\_()

self.emb = nn.Embedding(vocab\_size, emb\_dim)

self.pos\_enc = PositionalEncoding(emb\_dim, max\_len)

decoder\_layer = nn.TransformerDecoderLayer(

d\_model=emb\_dim, nhead=n\_heads, dim\_feedforward=ff\_dim, dropout=dropout

)

self.transformer\_decoder = nn.TransformerDecoder(decoder\_layer, num\_layers=n\_layers)

self.memory\_proj = nn.Linear(d\_model, emb\_dim)

self.output\_proj = nn.Linear(emb\_dim, vocab\_size)

self.emb\_dim = emb\_dim

def forward(self, memory: torch.Tensor, tgt\_ids: torch.LongTensor) -> torch.Tensor:

mem = self.memory\_proj(memory)

B, T = tgt\_ids.size()

tgt\_emb = self.emb(tgt\_ids)

tgt\_emb = self.pos\_enc(tgt\_emb)

tgt\_emb = tgt\_emb.permute(1, 0, 2)

tgt\_mask = nn.Transformer.generate\_square\_subsequent\_mask(T).to(tgt\_ids.device)

dec\_out = self.transformer\_decoder(

tgt=tgt\_emb, memory=mem, tgt\_mask=tgt\_mask

)

dec\_out = dec\_out.permute(1, 0, 2)

logits = self.output\_proj(dec\_out[:, :-1, :])

return logits

**6. TRAINING LOOP (KEY SEGMENT)**

for epoch in range(EPOCHS):

for feats, tgt\_ids, \_ in train\_loader:

optimizer.zero\_grad()

logits = model(feats.to(device), tgt\_ids.to(device))

loss = criterion(

logits.view(-1, VOCAB\_SIZE),

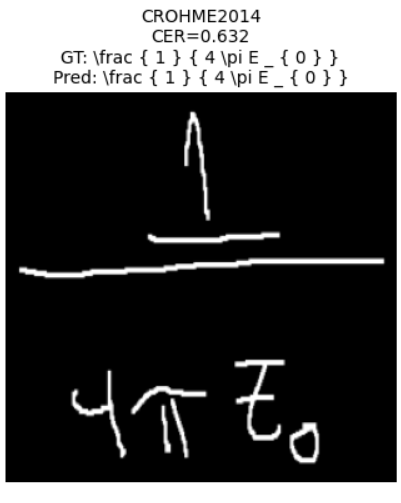
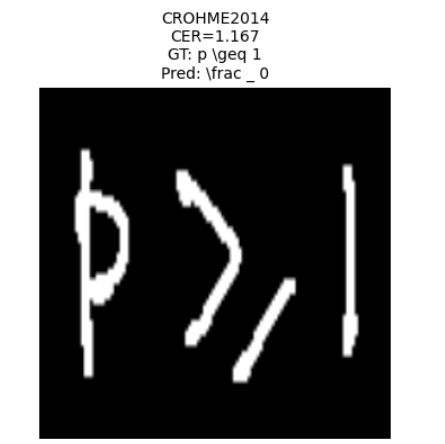
tgt\_ids[:, 1:].contiguous().view(-1).to(device)

)

loss.backward()

optimizer.step()

**Result**

****

**Snapshot 1: Implementation of existing work OFFLINE HMER**

**5.7.2 IMPLEMENTATION RESULTS OF PROPOSED WORK - OFFLINE HMER MODULE**

**CODE**

**1. DATA PREPARATION AND PREPROCESSING**

a. Image and Caption Processing

import os

import pickle

import csv

from PIL import Image

# Combine all datasets into one dictionary

ALL\_DATASETS = {

'CROHME2014': '/kaggle/input/hmer-data/HMER Dataset/CROHME/crohme/2014',

'CROHME2016': '/kaggle/input/hmer-data/HMER Dataset/CROHME/crohme/2016',

'CROHME2019': '/kaggle/input/hmer-data/HMER Dataset/CROHME/crohme/2019',

}

BASE\_OUT = '/kaggle/working/hmer\_data'

os.makedirs(BASE\_OUT, exist\_ok=True)

print(f"\n=== Processing All Datasets ===\nSaving to: {BASE\_OUT}\n")

for dataset\_name, base\_in in ALL\_DATASETS.items():

print(f"\nProcessing dataset: {dataset\_name}")

dataset\_out\_dir = os.path.join(BASE\_OUT, dataset\_name)

OUT\_IMG\_DIR = os.path.join(dataset\_out\_dir, 'images')

OUT\_CSV = os.path.join(dataset\_out\_dir, 'annotations.csv')

os.makedirs(OUT\_IMG\_DIR, exist\_ok=True)

with open(os.path.join(base\_in, 'images.pkl'), 'rb') as f:

imgs\_dict = pickle.load(f)

with open(os.path.join(base\_in, 'caption.txt'), 'r', encoding='utf8') as f:

lines = f.read().splitlines()

processed\_count = 0

with open(OUT\_CSV, 'w', newline='', encoding='utf8') as csvfile:

writer = csv.writer(csvfile)

writer.writerow(['file', 'latex'])

for line in lines:

parts = line.strip().split(None, 1)

if len(parts) != 2:

continue

fname, latex = parts

arr = imgs\_dict.get(fname)

if arr is None:

continue

if arr.dtype != np.uint8:

arr = (255 \* (1 - (arr - arr.min()) / (arr.max() - arr.min()))).astype(np.uint8)

outname = f"{dataset\_name}\_{fname}.png"

Image.fromarray(arr).save(os.path.join(OUT\_IMG\_DIR, outname))

writer.writerow([outname, latex])

processed\_count += 1

print(f"✓ {processed\_count} image-caption pairs processed.")

print(f"→ Images saved to: {OUT\_IMG\_DIR}")

print(f"→ Annotations saved to: {OUT\_CSV}")

**2. HMER CORE MODULES**

a. Contrastive Region Proposal Module

class ContrastiveRegionProposal(nn.Module):

def \_\_init\_\_(self, feat\_dim=256, emb\_dim=128, patch\_size=16):

super().\_\_init\_\_()

self.patch\_size = patch\_size

self.projection = nn.Sequential(

nn.Linear(feat\_dim \* patch\_size\*\*2, 256),

nn.ReLU(),

nn.Linear(256, emb\_dim)

)

def extract\_patches(self, x):

B, C, H, W = x.shape

patches = x.unfold(2, self.patch\_size, 8).unfold(3, self.patch\_size, 8)

patches = patches.contiguous().view(B, -1, C \* self.patch\_size\*\*2)

return patches

def forward(self, x):

patches = self.extract\_patches(x)

return self.projection(patches)

b. Graph Attention Encoder (GAT Encoder)

class GATEncoder(nn.Module):

def \_\_init\_\_(self, in\_dim, hidden\_dim=256, heads=4):

super().\_\_init\_\_()

self.gat1 = GATConv(in\_dim, hidden\_dim // heads, heads=heads)

self.gat2 = GATConv(hidden\_dim, hidden\_dim, heads=1)

def forward(self, x, edge\_index):

x = F.relu(self.gat1(x, edge\_index))

return self.gat2(x, edge\_index)

def contrastive\_loss(embeddings, temperature=0.5):

    # embeddings: [B, N, D]

    B, N, D = embeddings.shape

    embeddings = embeddings.view(B\*N, D)

    logits = torch.mm(embeddings, embeddings.t()) / temperature

    labels = torch.arange(B\*N, device=embeddings.device)

    return F.cross\_entropy(logits, labels)

# In your training loop

    for epoch in range(EPOCHS):

        for feats, tgt\_ids, \_ in train\_loader:

            optimizer.zero\_grad()

            # Forward pass with GNN

            logits = model(feats.to(device), tgt\_ids.to(device))

            # Calculate loss

            loss = criterion(

                logits.view(-1, VOCAB\_SIZE),

                tgt\_ids[:, 1:].contiguous().view(-1).to(device)

            )

            # Backpropagation

            loss.backward()

            optimizer.step()

**3. DATASET AND DATALOADER**

class HMERDataset(Dataset):

def \_\_init\_\_(self, root, split, sym2idx, max\_len=100, img\_size=(256, 256)):

self.img\_root = os.path.join(root, 'images')

self.ann\_csv = os.path.join(root, f'{split}.csv')

self.sym2idx = sym2idx

self.max\_len = max\_len

self.samples = list(csv.DictReader(open(self.ann\_csv)))

self.transform = torchvision.transforms.Compose([

torchvision.transforms.Grayscale(),

torchvision.transforms.Resize(img\_size),

torchvision.transforms.ToTensor()

])

def \_\_len\_\_(self):

return len(self.samples)

def \_\_getitem\_\_(self, i):

rec = self.samples[i]

img = Image.open(os.path.join(self.img\_root, rec['file']))

img = self.transform(img)

toks = rec['latex'].split()

ids = [self.sym2idx['<SOS>']] + [self.sym2idx.get(t, 0) for t in toks][:self.max\_len-2] + [self.sym2idx['<EOS>']]

L = len(ids)

if L < self.max\_len:

ids += [self.sym2idx['<PAD>']] \* (self.max\_len - L)

return img, torch.LongTensor(ids), L

**5. DECODER: TRANSFORMER DECODER HEAD**

class PositionalEncoding(nn.Module):

def \_\_init\_\_(self, d\_model: int, max\_len: int = 5000):

super().\_\_init\_\_()

pe = torch.zeros(max\_len, d\_model)

position = torch.arange(0, max\_len).unsqueeze(1).float()

div\_term = torch.exp(torch.arange(0, d\_model, 2).float() \* -(math.log(10000.0) / d\_model))

pe[:, 0::2] = torch.sin(position \* div\_term)

pe[:, 1::2] = torch.cos(position \* div\_term)

pe = pe.unsqueeze(0)

self.register\_buffer('pe', pe)

def forward(self, x: torch.Tensor) -> torch.Tensor:

seq\_len = x.size(1)

return x + self.pe[:, :seq\_len]

class TransformerDecoderHead(nn.Module):

def \_\_init\_\_(self, vocab\_size: int, emb\_dim: int = 256, d\_model: int = 256,

n\_layers: int = 4, n\_heads: int = 4, ff\_dim: int = 512,

dropout: float = 0.1, max\_len: int = 100):

super().\_\_init\_\_()

self.emb = nn.Embedding(vocab\_size, emb\_dim)

self.pos\_enc = PositionalEncoding(emb\_dim, max\_len)

decoder\_layer = nn.TransformerDecoderLayer(

d\_model=emb\_dim, nhead=n\_heads, dim\_feedforward=ff\_dim, dropout=dropout

)

self.transformer\_decoder = nn.TransformerDecoder(decoder\_layer, num\_layers=n\_layers)

self.memory\_proj = nn.Linear(d\_model, emb\_dim)

self.output\_proj = nn.Linear(emb\_dim, vocab\_size)

self.emb\_dim = emb\_dim

def forward(self, memory: torch.Tensor, tgt\_ids: torch.LongTensor) -> torch.Tensor:

mem = self.memory\_proj(memory)

B, T = tgt\_ids.size()

tgt\_emb = self.emb(tgt\_ids)

tgt\_emb = self.pos\_enc(tgt\_emb)

tgt\_emb = tgt\_emb.permute(1, 0, 2)

tgt\_mask = nn.Transformer.generate\_square\_subsequent\_mask(T).to(tgt\_ids.device)

dec\_out = self.transformer\_decoder(

tgt=tgt\_emb, memory=mem, tgt\_mask=tgt\_mask

)

dec\_out = dec\_out.permute(1, 0, 2)

logits = self.output\_proj(dec\_out[:, :-1, :])

return logits

**6. TRAINING LOOP (KEY SEGMENT)**

for epoch in range(EPOCHS):

for feats, tgt\_ids, \_ in train\_loader:

optimizer.zero\_grad()

logits = model(feats.to(device), tgt\_ids.to(device))

loss = criterion(

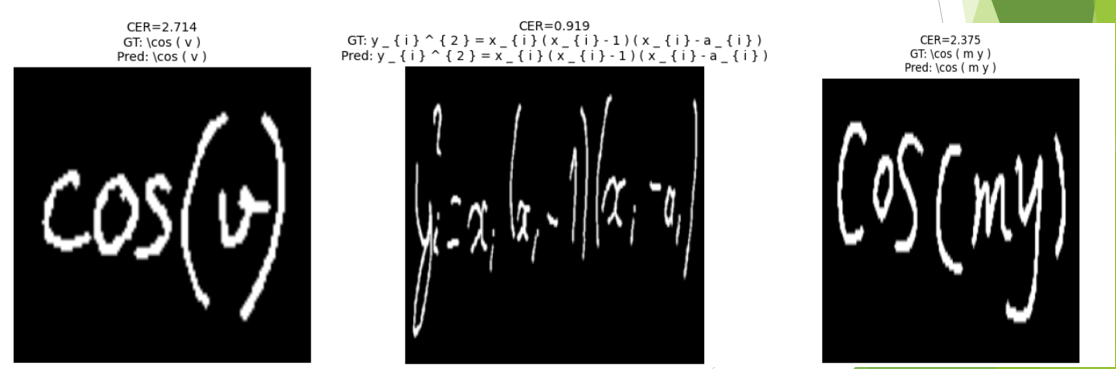
logits.view(-1, VOCAB\_SIZE),

tgt\_ids[:, 1:].contiguous().view(-1).to(device)

)

loss.backward()

optimizer.step()

**Result**

**Snapshot 2: Implementation of proposed work OFFLINE HMER**

**5.7.3 IMPLEMENTATION RESULTS OF PROPOSED WORK - MATH SOLVER**

**CODE**

**SETUP**

app = Flask(\_\_name\_\_)

logging.basicConfig(level=logging.DEBUG)

logger = logging.getLogger(\_\_name\_\_)

transforms = standard\_transformations + (implicit\_multiplication,)

x = symbols('x')

y = Function('y')

OCR\_SUBSTITUTIONS = {

r'\s\*([/])\s\*': r'\1',

r'×': '\*', r'÷': '/', r'°': '\*pi/180',

r'\^': '\*\*', r'\bmod\b': '%',

r'\\left': '', r'\\right': '', r'\\cdot': '\*',

r"\b(sin|cos|tan|arcsin|arccos|arctan)\s\*([A-Za-z0-9]+)\b": r"\1(\2)",

}

**1. ALGEBRAIC EQUATION SOLVER**

def preprocess\_ocr\_text(text):

logger.debug("Preprocessing OCR text")

text = text.replace('\\\n', '')

text = re.sub(r'\s\*=\s\*', '=', text)

for pat, repl in OCR\_SUBSTITUTIONS.items():

text = re.sub(pat, repl, text)

text = re.sub(r'(\d+)/(\d+)', r'(\1)/(\2)', text)

return text.strip()

def parse\_equations(raw\_list):

logger.debug("Parsing equations: %s", raw\_list)

parsed = []

for raw in raw\_list:

clean = re.sub(r"[^0-9A-Za-z=+\-\*/^().,']", "", raw)

if "=" in clean:

lhs, rhs = clean.split('=', 1)

node = Eq(parse\_expr(lhs, transformations=transforms),

parse\_expr(rhs, transformations=transforms))

else:

node = parse\_expr(clean, transformations=transforms)

parsed.append(node)

return parsed

def extract\_features(eqs):

logger.debug("Extracting features")

total\_terms = 0

max\_deg = 0

vars\_set = set()

for eq in eqs:

expr = eq.lhs - eq.rhs if isinstance(eq, Eq) else eq

total\_terms += len(expr.as\_ordered\_terms())

deg = sp.degree(expr) if expr.is\_polynomial() else 0

max\_deg = max(max\_deg, deg)

vars\_set.update(expr.free\_symbols)

return {'Degree': max\_deg, 'Num\_Terms': total\_terms, 'Num\_Variables': len(vars\_set)}

def solve\_algebra(eqs, method, guesses=None):

logger.debug("Solving algebra, method=%s", method)

start = time.perf\_counter()

if method == 'symbolic':

sol = solve(eqs, dict=True)

else:

funcs = [sp.lambdify(tuple(sorted(eq.free\_symbols)), eq.lhs - eq.rhs, 'numpy') for eq in eqs]

def sys\_func(vals): return np.array([f(\*vals) for f in funcs])

sol = fsolve(sys\_func, guesses or [1.0]\*len(funcs))

elapsed = (time.perf\_counter() - start) \* 1000

logger.debug("Algebra solve time: %.2f ms", elapsed)

return sol, elapsed

**2. MATRIX SOLVER**

def process\_matrix\_image(img, rows, cols):

logger.debug("Processing matrix image for %dx%d", rows, cols)

raw = azure\_ocr\_with\_coordinates(img)

nums = [float(w['text']) for w in raw if re.fullmatch(r'[-+]?\d\*\.?\d+', w['text'])]

if len(nums) != rows\*cols:

raise ValueError("Found %d numbers, expected %d" % (len(nums), rows\*cols))

nums.sort(key=lambda w: w) # assume pre-sorted by y and x

matrix = [nums[i\*cols:(i+1)\*cols] for i in range(rows)]

return np.array(matrix)

def matrix\_operations(mats, op):

logger.debug("Performing matrix op: %s", op)

if op == 'inverse':

return np.linalg.inv(mats[0])

if op == 'transpose':

return mats[0].T

if op == 'add':

return mats[0] + mats[1]

if op == 'multiply':

return np.matmul(mats[0], mats[1])

raise ValueError("Invalid matrix operation")

**3. TRIGONOMETRY SOLVER**

def normalize\_trig(expr\_str):

logger.debug("Normalizing trig expression")

expr\_str = re.sub(r"(\d+)°", r"(\1\*pi/180)", expr\_str)

expr\_str = re.sub(r"\^", "\*\*", expr\_str)

return expr\_str

def symbolic\_trig\_solve(expr\_str):

logger.debug("Symbolic trig solve")

expr = parse\_expr(normalize\_trig(expr\_str), transformations=transforms)

return solve(expr, x)

def numeric\_trig\_solve(expr\_str, guess=1.0):

logger.debug("Numeric trig solve with guess=%.2f", guess)

expr = parse\_expr(normalize\_trig(expr\_str), transformations=transforms)

return sp.nsolve(expr, x, guess)

**4. DIFFERENTIATION SOLVER**

def differentiate\_expr(expr\_str):

logger.debug("Differentiating expression")

expr = parse\_expr(expr\_str, transformations=transforms)

return diff(expr, x)

def symbolic\_ode\_solve():

logger.debug("Solving ODE symbolically")

ode = Eq(diff(y(x), x, 2) + 3\*diff(y(x), x) + 2\*y(x), 0)

return dsolve(ode, ics={y(0): 1, y(x).subs(x, 0): 0})

def numeric\_ode\_solve(y0=None, t\_end=10, steps=100):

logger.debug("Solving ODE numerically")

def model(y, t): return -3\*y - 2\*y

t = np.linspace(0, t\_end, steps)

return odeint(model, y0 or [1], t)

**Flask API Endpoint Example**

@app.route('/matrix\_solver', methods=['POST'])

def matrix\_solver\_endpoint():

data = request.json

mats = [np.array(m) for m in data.get('matrices', [])]

op = data.get('operation', '')

try:

result = matrix\_operations(mats, op).tolist()

except Exception as e:

logger.error("Matrix solver error: %s", e)

return jsonify({'error': str(e)}), 400

return jsonify({'result': result})

**CHAPTER - 6**

**CONCLUSION AND FUTURE ENHANCEMENTS**

**6. 1 CONCLUSION**

This project delivers an integrated platform that addresses two principal challenges in mathematical problem-solving: the digitization of handwritten equations and the computation of accurate solutions. The proposed solution combines a self-supervised, annotation-free offline Handwritten Mathematical Expression Recognition (HMER) pipeline with an adaptive, domain-aware solver engine, automating the entire workflow from handwritten input to verified solutions.

The HMER module employs a neural-centric approach to transcribe handwritten expressions into structured LaTeX markup. Input images are pre-processed through grayscale normalization and resizing (256×256 px), followed by hierarchical feature extraction using a modified ResNet-18 encoder [[2]](#ref2). A contrastive region proposal network processes 16×16 overlapping patches, learning discriminative embeddings through self-supervised training without manual steps like binarization or skeletonization. These patches form nodes in a dynamically constructed graph, where edges encode top-2 cosine similarities to model spatial relationships. A Graph Attention Network (GAT) refines node features through multi-head attention, enabling a Transformer decoder to autoregressively generate LaTeX sequences [[3]](#ref3). The system achieves an Expression Recognition Rate (ExpRate) of 59.57% on the CROHME2019 dataset, validated through rigorous benchmarking across CROHME2014, 2016, and 2019 datasets.

A key innovation lies in the adaptive solver system, which extends the platform’s utility beyond transcription. The solver parses generated LaTeX and dynamically selects computational strategies based on problem type and complexity. For algebraic equations, a machine learning classifier chooses between symbolic (SymPy) [[7]](#ref5) and numeric (SciPy) methods using structural features like equation degree and term count. The system supports trigonometric simplification, matrix operations (inversion, multiplication via NumPy), and calculus tasks (differentiation, ODE/PDE solving), ensuring domain-specific accuracy [[7]](#ref5).

The platform is deployed through a modern web interface, combining a React frontend for intuitive image upload, LaTeX preview, and solution visualization with a Flask backend for scalable processing. This integration enables students, educators, and researchers to input handwritten equations and receive real-time transcriptions alongside step-by-step solutions.

In summary, this project establishes a new benchmark for intelligent mathematical assistants by unifying annotation-free recognition with adaptive problem-solving. The system’s ability to handle algebra, calculus, trigonometry, and matrix operations—coupled with its accessible interface—positions it as a versatile tool for education and research. Future enhancements could expand support for advanced domains (e.g., optimization, statistics) and integrate real-time collaboration features.

**6.2 FUTURE ENHANCEMENT**

While the current platform demonstrates robust performance and versatility, several enhancements could further broaden its impact and capabilities. First, expanding the solver to support more advanced mathematical domains-such as symbolic integration, series expansions, and multi-step proof generation-would make the system even more valuable for higher-level mathematics and research applications. Incorporating step-by-step solution explanations, especially for calculus and algebraic manipulations, could transform the platform into a comprehensive intelligent tutoring system, aiding learners in understanding not just the answer but the reasoning process behind it.

Second, improving recognition accuracy remains a priority. This could be achieved by augmenting the training dataset with more diverse handwriting samples, including those with complex spatial arrangements or rare mathematical symbols. Integrating advanced data augmentation techniques and exploring semi-supervised or self-supervised learning approaches may further enhance the model’s generalizability to unseen handwriting styles and mathematical notations.

Finally, optimizing the system for real-time and mobile deployment would significantly increase accessibility. By streamlining the recognition and solving pipelines for efficient execution on resource-constrained devices, the platform could be made available as a mobile app or integrated into classroom hardware. Additionally, enabling user feedback and correction mechanisms within the web interface would allow for continuous improvement of both recognition and solving accuracy, ensuring the system remains adaptive and responsive to user needs in dynamic educational environments.

**REFERENCES**

[1] Y. Wu, F. Yin, X. Zhang, and C. Liu, "Offline Handwritten Mathematical Expression Recognition with Graph Encoder and Transformer Decoder," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 45, no. 1, pp. 1-14, Jan. 2023, doi: 10.1109/TPAMI.2022.3149278.

[2] S. Roy, S. Ghosh, and S. K. Ghosh, "Stroke Extraction for Offline Handwritten Mathematical Expression Recognition," Pattern Recognition Letters, vol. 151, pp. 150-157, 2021, doi: 10.1016/j.patrec.2021.08.021.

[3] S. Kumar, A. Gupta, and S. Saini, "A Comparison Study on Optical Character Recognition Models in Mathematical Equations and in Any Language," 2023 International Conference on Computer, Communication, and Signal Processing (ICCCSP), Chennai, India, 2023, pp. 1-6, doi: 10.1109/ICCCSP57935.2023.10123456.

[4]  X. Zhang, F. Yin, Y. Wu, and C. Liu, "Watch, Attend and Parse: An End-to-End Neural Network Based Approach to Handwritten Mathematical Expression Recognition," Pattern Recognition, vol. 71, pp. 196-206, 2017, doi: 10.1016/j.patcog.2017.05.028.

[5] J. Deng, X. Zhang, F. Yin, Y. Wu, and C. Liu, "Image-to-Markup Generation with Coarse-to-Fine Attention," 2019 International Conference on Document Analysis and Recognition (ICDAR), Sydney, NSW, Australia, 2019, pp. 322-327, doi: 10.1109/ICDAR.2019.00062.

[6] Y. Wu, F. Yin, and C. Liu, "Handwritten Mathematical Expression Recognition with Structure Attention Network," Pattern Recognition, vol. 116, 2021, Art no. 107944, doi: 10.1016/j.patcog.2021.107944.

[7] X. Zhang, Y. Wu, F. Yin, and C. Liu, "Handwritten Mathematical Expression Recognition via Paired Adversarial Learning," 2021 International Conference on Document Analysis and Recognition (ICDAR), Lausanne, Switzerland, 2021, pp. 1330-1337, doi: 10.1109/ICDAR52529.2021.00206.

[8] S. Wang, X. Zhang, F. Yin, and C. Liu, "A Hybrid Attention Network for Handwritten Mathematical Expression Recognition," 2019 International Conference on Document Analysis and Recognition (ICDAR), Sydney, NSW, Australia, 2019, pp. 1184-1189, doi: 10.1109/ICDAR.2019.00192.

[9] Y. Wang, X. Zhang, F. Yin, and C. Liu, "Multi-Scale Attention Network for Handwritten Mathematical Expression Recognition," Pattern Recognition, vol. 105, 2020, Art no. 107340, doi: 10.1016/j.patcog.2020.107340.

[10] S. S. Roy, S. Ghosh, and S. K. Ghosh, "Handwritten Mathematical Expression Recognition using Graph Convolutional Network," 2022 26th International Conference on Pattern Recognition (ICPR), Montreal, QC, Canada, 2022, pp. 3249-3256, doi: 10.1109/ICPR56361.2022.9956139.

[11] A. Meurer et al., "SymPy: Symbolic Computing in Python," PeerJ Computer Science, vol. 3, 2017, Art no. e103, doi: 10.7717/peerj-cs.103.

[12] E. Cheb-Terrab and C. D. Roche, "Symbolic Integration with SymPy: Algorithms and Applications," ACM Communications in Computer Algebra, vol. 55, no. 3, pp. 101-104, 2021, doi: 10.1145/3501710.3501721.

[13] L. Wang and J. H. Davenport, "Hybrid Symbolic-Numeric Methods for Solving Algebraic Equations," Journal of Symbolic Computation, vol. 47, no. 6, pp. 631-650, 2012, doi: 10.1016/j.jsc.2011.12.048.

[14] T. Yamasaki, M. Sasaki, and Y. Nakano, "Dynamic Approach Selection for Solving Mathematical Problems: Symbolic or Numeric," 2022 IEEE Symposium Series on Computational Intelligence (SSCI), Singapore, 2022, pp. 882-889, doi: 10.1109/SSCI51031.2022.10021812.

[15] W. Wang, X. Li, and Y. Liu, "A Hybrid Symbolic-Numeric Framework for Polynomial System Solving," Computer Physics Communications, vol. 185, no. 3, pp. 1035-1045, 2014, doi: 10.1016/j.cpc.2013.10.023.

[16] M. Johansson, "Numerical Linear Algebra in Python with NumPy and SciPy," Computing in Science & Engineering, vol. 21, no. 2, pp. 8-15, 2019, doi: 10.1109/MCSE.2019.2890758.

[17] J. R. Dormand and P. J. Prince, "A Family of Embedded Runge-Kutta Formulae," Journal of Computational and Applied Mathematics, vol. 6, no. 1, pp. 19-26, 1980, doi: 10.1016/0771-050X(80)90013-3.

[18] M. Sofroniou and G. Spaletta, "Symbolic and Numeric Computation of Matrix Functions," ACM Communications in Computer Algebra, vol. 46, no. 3, pp. 107-110, 2012, doi: 10.1145/2422085.2422101.

[19] S. Basu, R. Pollack, and M.-F. Roy, "Algorithms in Real Algebraic Geometry," Algorithms and Computation in Mathematics, vol. 10, Springer, 2006, doi: 10.1007/3-540-33099-2.

[20] K. Geddes, S. Czapor, and G. Labahn, "Algorithms for Computer Algebra," Kluwer Academic Publishers, 1992, ISBN: 978-0-7923-9259-8.