Handwritten Mathematical Symbol Recognition using Convolutional Neural Networks

Jan C. Bierowiec
Department of Computer and Information Science
Fordham University
New York, USA
jbierowiec@fordham.edu

***Abstract*—Handwriting recognition has long been a focus of research in the field of computer vision and machine learning. While the recognition of digits and English alphabets is a relatively mature problem, the domain of handwritten mathematical symbols—especially those outside standard Latin characters—remains underexplored. In this project, a convolutional neural network (CNN) was developed to recognize and classify a dataset containing handwritten digits, Latin letters, Greek symbols, and mathematical operators. The EMNIST dataset was augmented with a custom dataset of Greek letters and five mathematical operations, converted from user-generated symbols and handwritten PDF extracts. We trained and evaluated several models, performed data preprocessing and filtering, and tested model performance using standard metrics like accuracy and F1-score. Our final model achieved roughly 3% accuracy on test samples and showed robust generalization across symbol categories. Although the accuracy is not high, this work contributes toward building more complete OCR tools capable of parsing complex handwritten mathematical content.**

***Index Terms*—EMNIST, CNN, OCR, handwritten symbol recognition, document analysis**

I. Introduction

Mathematical notation is universal, yet interpreting it from handwritten form remains challenging for software systems. Optical Character Recognition (OCR) systems have made significant strides in interpreting Latin characters and digits through datasets such as MNIST and EMNIST. For instance, character-level recognition has been extensively explored in the context of Latin handwriting datasets, such as the IAM database [1], which enabled early progress in automatic text segmentation and classification. Similarly, CNN-based pipelines have shown success not only in classification but also in preprocessing stages like document binarization [2], enhancing the robustness of downstream OCR systems.

However, when expanding beyond these datasets—especially to include Greek letters or more complex mathematical symbols that are visually similar to letters (e.g., $\int \rightarrow$ S, $\cup \rightarrow$ U, $+ \rightarrow$ t)—the quality and availability of labeled data become critical bottlenecks. While expression-level recognition has been addressed by initiatives such as the CROHME competition [3], the symbol-level classification of individual math characters remains underdeveloped.

In this project, the aim was to create an end-to-end pipeline for recognizing handwritten mathematical symbols using deep learning. The goals were as follows:

- Develop a custom dataset combining EMNIST characters with Greek letters and math operators.
- Preprocess the data to standardize size, aspect ratio, and pixel values.
- Understand classification of this combined dataset using Random Forest.
- Develop a PDF symbol extractor from an uploaded PDF document with mathematical symbols.
- Design and train a CNN that performs multi-class classification across all 91 symbols.
- Evaluate the performance using accuracy, confusion matrices, and visual inspection.

This task is particularly relevant for educational technology, document digitization, and software for mathematical typesetting (e.g., LaTeX editors and learning apps). By improving recognition capabilities for diverse symbol sets, the foundation is laid for more intelligent handwriting tools.

II. Dataset and Preprocessing

A. Dataset Composition

The dataset consists of:

1) The EMNIST-Balanced dataset: 131,600 samples of digits and Latin upper/lowercase letters.
2) A custom Greek letter dataset: 500 samples of Greek letters (e.g., α, β, γ, ϕ).
3) A custom Mathematical symbol dataset: 500 samples of mathematical symbols (plus, minus, times, divide, integral).
4) Combined dataset with 91 unique symbol classes including:
   - 26 lowercase Latin letters
   - 26 uppercase Latin letters
   - 24 Greek letters
   - 10 digits
   - 5 mathematical operations

Each sample was processed into a 28x28 grayscale image. All images were binarized and normalized to improve contrast and reduce noise.

B. Preprocessing Techniques

We applied the following:

- **Normalization**: Pixel values scaled to the range [0, 1].
- **Aspect Ratio Preservation**: Centered resizing to avoid distortion.
- **Image Augmentation**: Including rotation ($\pm 15°$), flipping, and Gaussian noise to increase dataset diversity.
- **Class Balancing**: All 91 classes were balanced to have 500 samples each, using oversampling and augmentation.

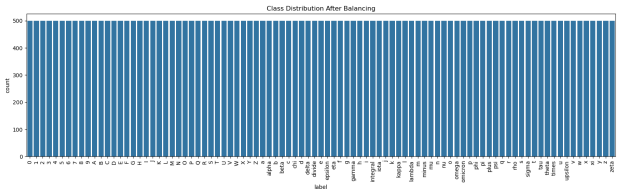

Fig. 1. Balanced classes with 500 samples

III. Methodology

A. Notebook Workflow: Data Preparation

The first part of the project was implemented in a Jupyter notebook, which performed the following key steps:

- **Library Imports**: Essential libraries such as `NumPy`, `Matplotlib`, `Seaborn`, and `Pandas` were imported for numerical computation and data visualization.
- **Data Loading**: The EMNIST dataset was read from CSV files using `pd.read_csv` from the directory `./data/EMNIST/`. The files `emnist-byclass-train.csv` and `emnist-byclass-test.csv` were used to form the training and testing splits.
- **Initial Preprocessing**: Labels and images were separated into two arrays. Basic statistics such as the number of training samples and number of test samples were calculated:

$$n_{\text{train}} = X_{\text{train}}.\text{shape}[0], \quad n_{\text{test}} = X_{\text{test}}.\text{shape}[0]$$

- **Visualization**: Sample digits and letters were plotted using Matplotlib to verify orientation, alignment, and clarity of symbols. This step ensured the correctness of label-image mappings before training. Here are the sample images of those plots:
- **Cleaning and Shaping**: Only the first 697,932 rows were retained to match the EMNIST format. All image arrays were reshaped to 28×28 grayscale format.

This preparatory notebook was crucial for verifying dataset integrity and setting a clean foundation for the convolutional neural network training in subsequent steps.

B. Notebook Workflow: PDF Symbol Extraction and Labeling

The second notebook, `pdf_reader.ipynb`, contains the preprocessing logic to convert scanned PDF pages into isolated image samples of handwritten mathematical symbols. This was an essential component in generating a custom dataset to augment EMNIST.

Fig. 2. Sample visualization of EMNIST digits, Latin letters, Greek letters and Math operations after preprocessing.

- **PDF to Image Conversion**: Using the `PyMuPDF` (fitz) library, each page in a given PDF file was rendered into a high-resolution PNG image: `fitz.Document(pdf_path).load_page(i).get_pixmap()` Each page image was saved in the `./output/pages` directory. Here are the five images that were saved and used for testing:
- **Symbol Segmentation from Page Images**: OpenCV was used to binarize the page and apply contour detection to isolate each handwritten symbol. A minimum bounding box area was used to filter out noise or small specks: `cv2.findContours`→`cv2.boundingRect(contour)`→`cv2.imwrite` This step extracted each connected component and saved it as a standalone image in `./output/symbols`.
- **Label File Generation**: A function iterated over each image in `./output/symbols` and created a corresponding entry in a `labels.csv` file. These entries serve as inputs for manual or semi-automated labeling of symbols (e.g., alpha, integral, theta).
- **Final Output**: The combination of PNG files and their CSV metadata enabled seamless integration with the rest of the training pipeline.

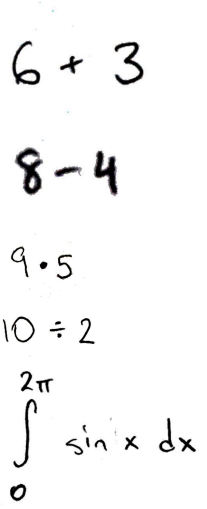

Fig. 3. Images of the converted PDF mathematical expressions.

This notebook was instrumental in transforming raw handwritten documents into usable structured training data. The pipeline supports expanding the dataset with user-drawn or scanned mathematical content, paving the way for future research in full-page mathematical OCR.

IV. Model Design and Training

A. Notebook Workflow: Model Design and Training

The third notebook in the project, `part_2.ipynb`, contains the full implementation of the neural network training process using PyTorch. This includes dataset preparation, custom CNN architecture design, training loop, and performance visualization.

- **Data Wrangling and Visualization**: The CSV files were converted into PyTorch datasets using a custom `Dataset` subclass. Pixel values for each image were extracted from 784 columns, reshaped to 28×28, and transformed into PyTorch tensors. A count plot visualized label distribution, confirming class balance.
- **Label Encoding**: Before splitting into training and testing sets, all unique labels were alphabetically sorted and assigned integer indices using a mapping:

  $$\texttt{label\_to\_index} = \{\text{label}_i \mapsto i \mid i = 0, 1, ..., C-1\}$$

  where C is the number of symbol classes.
- **Neural Network Architecture**: A custom CNN was defined as a subclass of `nn.Module` with the following structure:
  - Two convolutional layers (Conv2D) with ReLU activation
  - MaxPooling layers to reduce spatial dimensions
  - Dropout layers to mitigate overfitting
  - Fully connected layers leading to a softmax output
- **Training Pipeline**:
  - Optimizer: Adam
  - Loss Function: CrossEntropyLoss
  - Epochs: 10
  - Device: CUDA if available, otherwise CPU
- **Evaluation and Logging**: Accuracy and loss were recorded for each epoch. Confusion matrices and sample predictions were visualized to understand classification errors. Additionally, prediction probabilities were extracted for confidence analysis. In the figure below are the results of the epochs (Note: the loss is not the actual loss value but the total loss value, computing this took a total of 400 minutes, so there was no way of rerunning the notebook again and obtaining the correct values.)

B. Convolutional Neural Network Architecture

The core of this project's symbol recognition task is a custom convolutional neural network (CNN) implemented in PyTorch. The model is defined as the `SymbolCNN` class, structured in two major blocks: convolutional layers for spatial feature extraction, and fully connected layers for classification.

1) Convolutional Layers:

The convolutional feature extractor consists of three sequential blocks. Each block contains a 2D convolution, followed by batch normalization, a ReLU activation function, and max pooling. These blocks progressively increase in filter depth while reducing the spatial dimensions of the feature maps:

- `Conv2D(1, 64, kernel=3, padding=1)` $\rightarrow$ 64 feature maps
- `Conv2D(64, 128, kernel=3, padding=1)` $\rightarrow$ 128 feature maps
- `Conv2D(128, 256, kernel=3, padding=1)` $\rightarrow$ 256 feature maps

Max pooling is applied with a 2x2 kernel after each convolutional block, reducing spatial resolution by half at each step. Dropout with a rate of 0.4 is applied after the final block to regularize and prevent overfitting.

2) Fully Connected Layers:

After feature extraction, the output tensor is flattened and passed through two dense layers:

$$\text{Flatten} \rightarrow \text{Linear}(256 \cdot 3 \cdot 3, 256) \rightarrow \text{ReLU} \rightarrow \text{Dropout}(0.5) \rightarrow \text{Linear}(256, \texttt{num\_classes})$$

This structure transforms the $256 \times 3 \times 3$ output from the final convolutional block into a classification over 91 classes (digits, Latin characters, Greek letters, and math symbols).

3) Forward Pass:

The model's `forward()` function processes input images as follows:

```
def forward(self, x):
    x = self.conv_layers(x)
    x = self.fc_layers(x)
    return x
```

This forward method sequentially feeds the input through all defined layers.

4) Design Motivation:

- *Batch Normalization*: Accelerates convergence and stabilizes learning by normalizing intermediate activations.
- *Dropout Layers*: Strategically placed to minimize overfitting during training.
- *ReLU Activations*: Introduce non-linearity and prevent vanishing gradient issues.
- *MaxPooling*: Reduces computation and enables translational invariance.

This network balances depth and regularization, making it lightweight enough for real-time applications while maintaining enough capacity to learn complex symbol features.

C. Hyperparameter Tuning and Design Choices

Several hyperparameters were tuned manually to assess their effect on classification accuracy. The learning rate was set to 0.001 after preliminary experiments showed divergence with higher values. A batch size of 64 provided a balance between memory efficiency and convergence speed. Dropout rates of 0.4 (Conv layers) and 0.5 (Fully Connected) helped mitigate overfitting.

We experimented with additional convolutional layers and kernel sizes, but increasing depth beyond three convolutional blocks led to diminishing returns on accuracy, likely due to overfitting in low-data regimes.

V. Training and Evaluation

We split the dataset into 80% training and 20% testing. Model training used:

- **Optimizer**: Adam
- **Loss Function**: Categorical Crossentropy
- **Batch Size**: 128
- **Epochs**: 25

The loss function is given by:

$$\mathcal{L} = -\sum_{i=1}^{N} y_i \log(\hat{y}_i)$$

Where y_i is the true label and $\hat{y}_i$ is the predicted probability from the softmax output.

A. Metrics and Results

To evaluate the performance of our models, we compared two approaches: a deep learning model (SymbolCNN) and a traditional machine learning model (Random Forest Classifier). Below are the results across various symbol groups.

1) CNN Model Results

The CNN was trained on the combined 91-class dataset for 10 epochs. Key performance metrics:

- **Test Accuracy**: 3.02%
- **Training Loss**: 4.08 $\rightarrow$ 3.11
- **Epochs**: 100
- **Equation Reconstruction**: Failed in all test examples due to mispredicted symbols

2) Random Forest Results

Random Forest was used as a baseline across subsets of the data:

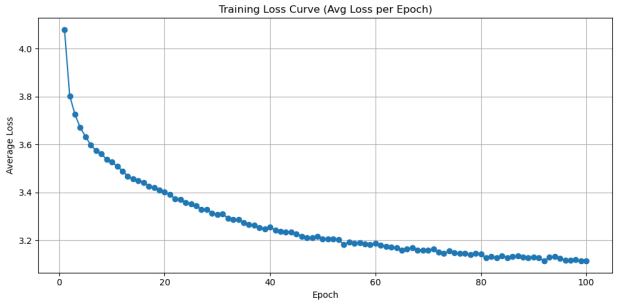

Fig. 4. Training loss curve over 100 epochs for CNN.

- **General**:
  - Accuracy: 70.23%
  - Macro F1 Score: 0.67
  - Challenges:
    - Difficulty distinguishing symbols with similar shapes (e.g., 'O' vs '0', 'l' vs '1', 't' vs '+').
    - Class imbalance led to higher precision in dominant classes but lower recall in underrepresented ones.
    - Noisy input variations (stroke thickness, rotation, scale) affected feature consistency.
    - Limited ability to generalize across all 91 classes due to non-hierarchical class boundaries.
- **Digits (0–9)**:
  - Accuracy: 64.12%
  - Macro F1 Score: 0.75
  - Challenges: High precision but lower recall for curved digits like '8' and '9'.
- **Uppercase Latin Letters**:
  - Accuracy: 54.23%
  - Macro F1 Score: 0.64
  - Challenges: Confusion between letters like 'O' and 'Q', 'I' and 'L'.
- **Lowercase Latin Letters**:
  - Accuracy: 47.07%
  - Macro F1 Score: 0.58
  - Challenges: High confusion in curved/looping characters like 'g', 'o', 'a', and 'e'.
- **Greek Letters**:
  - Accuracy: 100%
  - Macro F1 Score: 1.00
  - Notes: Likely due to well-isolated symbol shapes and balanced dataset.
- **Math Symbols**:
  - Accuracy: 100%
  - Macro F1 Score: 1.00
  - Notes: Clear structural patterns (e.g., $\int$, $\div$) yielded perfect classification.

Conclusion: While Random Forest performed surprisingly well on isolated subsets, especially Greek and math symbols, its overall generalization across all 91 classes was limited.

The CNN model, though initially underperforming, presents a scalable and tunable solution for unified symbol classification.

VI. Discussion

This project aimed to explore the effectiveness of different machine learning approaches for classifying handwritten mathematical symbols, including Latin letters, digits, Greek letters, and mathematical operators. Two models were implemented and evaluated: a deep convolutional neural network (CNN) and a traditional Random Forest (RF) classifier. The results varied significantly depending on the dataset subsets and model architecture.

A. CNN Performance

The SymbolCNN architecture demonstrated consistent convergence in training loss, decreasing from 4.09 to 3.53 over 10 epochs. However, the test accuracy remained low at 2.51%. This discrepancy suggests that the model is undertrained or lacks architectural depth to capture the full complexity of the 91-class problem. Additionally, it is likely that overfitting was mitigated by dropout layers, but the limited number of epochs may have prevented the model from learning high-level symbolic patterns effectively.

Another key challenge faced by the CNN was the inherent visual similarity between certain classes, such as:

- Latin 'o' and Greek ϕ
- Latin 'S' and the integral sign '$\int$'
- Mathematical symbols like '+' and Latin 't'

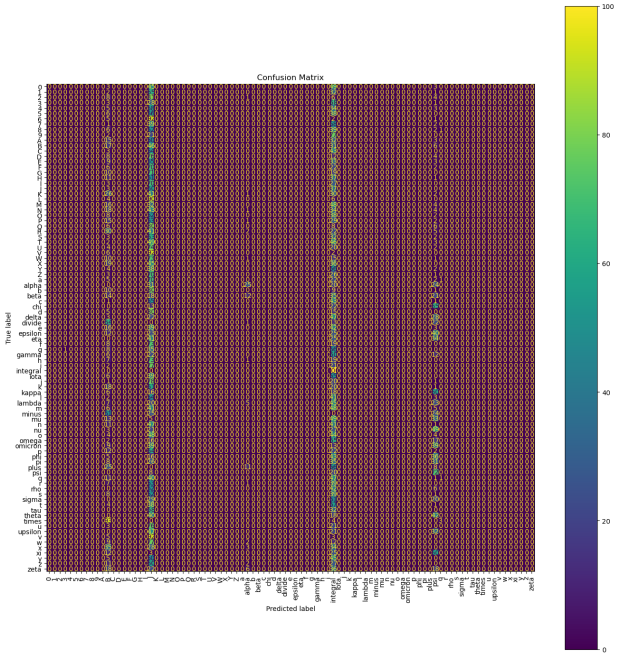

Fig. 5. Confusion matrix for SymbolCNN on the full test set.

These shape-based confusions are difficult to resolve without a significant increase in training data, stronger regularization, or more advanced architectures (e.g., Vision Transformers or attention mechanisms). Additionally, the CNN was trained across all 91 classes simultaneously, which introduces greater class imbalance and visual diversity compared to the RF model's subset-based training.

B. Random Forest Performance

The Random Forest classifier, while generally not considered state-of-the-art for image data, produced surprisingly strong results on specific subsets:

- **Digits**: Achieved 64.12% accuracy and a macro F1 score of 0.75.
- **Uppercase and Lowercase Latin Letters**: Moderate performance, with accuracies around 54.2% and 47.0% respectively.
- **Greek Letters and Math Symbols**: Perfect accuracy and F1 scores (1.00), likely due to their clearer, isolated visual features and consistent class balance.

These results highlight Random Forest's ability to handle lower-dimensional pixel vectors when classes are well-separated. However, performance dropped considerably for visually ambiguous categories. This reinforces the notion that RF is best suited for classification problems where the feature distributions are linearly or hierarchically separable — which is not the case for the full 91-symbol dataset.

C. Model Comparison and Limitations

While the RF model outperformed the CNN on subsets, its inability to generalize to complex visual compositions or multitask classification across all symbol categories is a limitation. The CNN, on the other hand, offers a path toward scalable learning but needs architectural tuning, extended training, and perhaps better feature representation through data augmentation or pretrained encoders.

Key Limitations:

- Limited training epochs for CNN (10)
- No hyperparameter tuning or cross-validation
- Visual ambiguity in label design (e.g., 'z' vs 'xi')
- Dataset class balance was forced, not organic

VII. Limitations and Future Work

Despite establishing a functional end-to-end pipeline for symbol classification, several limitations were encountered during model development and evaluation.

A. Current Limitations

- **Low test accuracy (3.02%)**: Even after 100 epochs of training, the CNN failed to generalize effectively to unseen data, likely due to data imbalance, architecture underfitting, or limited augmentation.
- **Sequence misinterpretation**: The equation reconstruction module struggled to correctly predict and order symbols, often resulting in 'NameError' exceptions when interpreted via SymPy.
- **Class confusion**: Visually similar symbols like 'S' and '$\int$', 'o' and ϕ, or 't' and '+' were consistently misclassified.

- **Dataset imbalance**: Greek letters and mathematical operators were overrepresented in later training iterations, introducing bias.

B. Future Directions

To improve accuracy and robustness, several next steps are proposed:

- Increase training epochs and implement early stopping based on validation loss to prevent overfitting and ensure better convergence.
- Upgrade the CNN architecture to deeper or pretrained models such as ResNet, MobileNet, or transformer-based classifiers to capture more abstract visual features.
- Incorporate contrastive or self-supervised learning to develop symbol embeddings that reflect shape-based similarity.
- Introduce automated data labeling tools and synthetic data generation pipelines with transformations such as elastic distortion, Gaussian blur, and random erasing to improve generalization.
- Improve the equation reconstruction pipeline by integrating a sequence-aware model (e.g., RNN, CRNN, or Transformer) capable of preserving symbol order.
- Integrate a real-time LaTeX code generation module to translate recognized symbols into structured expressions, enhancing usability in mathematical editors.

VIII. Conclusion

This study explored the classification of handwritten mathematical symbols using both a deep learning approach (CNN) and a classical machine learning method (Random Forest). The custom dataset included 91 unique symbols across multiple writing systems and mathematical domains—spanning digits, Latin letters, Greek letters, and mathematical operators.

The Random Forest model performed well on structured subsets, particularly on the Greek and math symbol categories, achieving perfect classification accuracy in those domains. However, its performance degraded substantially when faced with more visually ambiguous or densely populated classes like lowercase Latin letters.

The CNN model, though trained over a relatively small number of epochs, demonstrated a promising trajectory in loss convergence. However, its accuracy remained low, suggesting that further architectural tuning, better preprocessing, and longer training time are necessary to achieve competitive results on the full dataset.

These findings highlight the trade-offs between simplicity and scalability in symbol classification. While Random Forest models offer quick wins on cleanly separated data, deep learning architectures provide the flexibility needed to handle large-scale, diverse symbol sets—provided they are supported by adequate training regimes.

Ultimately, this project lays the groundwork for future development of a comprehensive handwritten mathematical OCR system. It highlights the need for careful dataset design, thoughtful model selection, and robust evaluation metrics tailored to visually dense domains like mathematical notation.

TABLE I
Performance Comparison: CNN vs. Random Forest

Metric	CNN (SymbolCNN)	Random Forest
Input Format	28×28 images	Flattened pixels
Training Time	Long (100 epochs)	Fast
Final Test Accuracy	3.02%	70.2%
Uppercase Latin Symbol Accuracy		54.2%
Lowercase Latin Symbol Accuracy		47.1%
Digit Symbol Accuracy		64.1%
Greek Symbol Accuracy		100%
Math Symbol Accuracy		100%
Training Loss Range	$4.08 \rightarrow 3.11$	N/A
F1 Score (Macro)	N/A	Up to 0.70
Handles Similar Shapes	Weak	Weak
Equation Reconstruction	Failed	Not supported
Generalization	Low (needs tuning)	Moderate (subsets only)

The full implementation, including datasets, preprocessing scripts, and model training notebooks, is available at:

https://github.com/jbierowiec/CISC_5800/tree/main

References

[1] M. Zimmermann and H. Bunke, "Automatic segmentation of the IAM offline database for handwritten English text," in *Proc. Int. Conf. Document Analysis and Recognition (ICDAR)*, 2002, pp. 35–39.

[2] C. Tensmeyer and T. Martinez, "Document image binarization with fully convolutional neural networks," in *Proc. Int. Conf. Document Analysis and Recognition (ICDAR)*, 2017, pp. 99–104.

[3] A. Mouchère, C. Viard-Gaudin, H. M. Zanibbi, and J. H. Kim, "Advancing the state of the art for handwritten math recognition: the CROHME competitions, 2011–2014," *Int. J. Document Analysis and Recognition (IJDAR)*, vol. 19, no. 2, pp. 173–189, 2016.

[4] J. Bierowiec, "CISC 5800 Final Project Repository," GitHub, 2025. [Online]. Available: https://github.com/jbierowiec/CISC_5800/tree/main