# HANDWRITTEN POLYNOMIAL EQUATION RECOGNITION USING CNN AND SYMBOLIC SOLUTION WITH SYMPY

**Anupa Gaire*, Rohisha Shrestha, Rosha Prajapati, Shristi Yakami, Avijit Karna**

Department of Computer Engineering, Khwopa Engineering College, Bhaktapur, Nepal

## Abstract

This study presents an easy-to-use system that can recognize and solve handwritten polynomial equations using a Convolutional Neural Network (CNN). To achieve this, we began by collecting digits and math symbols from various publicly available sources. We began with a dataset of around 30,000 samples, then converted the images to grayscale, inverted them, applied binary thresholding, and removed noise to clean them up. Individual symbols are then separated using OpenCV, and a custom-trained CNN model classifies each symbol. We used 16 classes to classify our data. To ensure the model's ability to detect symbols increases and produces efficient results, we expanded our data set through data augmentation to over $100,000$ images. CNN, built using Keras, achieved an impressive $98.99\%$ classification accuracy, reliably identifying each character during training. Once the symbols are recognized, they are combined to form a complete equation. For normal equations, the system uses the eval () function to evaluate expression, providing quick, dynamic solutions . For polynomial equations of up to the third degree, SymPy is utilized for symbolic computation, providing accurate and detailed solutions. The solution is presented through a PyQt5-based user interface, that allows users to upload their handwritten equations and view the solved results. The proposed method supports polynomial equations with variables and basic mathematical operators, providing an accurate and user-friendly educational tool. The system supports basic mathematical symbols $(+, -, \times, =, x, y)$. It can handle polynomial equations up to the third degree, which will be beneficial for students learning algebra or for quick problem solving.

**Keywords:** CNN, Handwritten equation solver, Imagep processing, PyQt5-interface, Polynomial recognition, Symbol segmentation

## 1. Introduction

Advancements in handwriting recognition and natural language processing have significantly improved machines' ability to interpret human input, resulting in more accurate and efficient tasks, such as image and text recognition. Over time, input methods have evolved from punched cards to keyboards, and now to touchscreens, voice commands, and even image-based recognition. With these advancements, solving mathematical problems has also become more convenient, as computers can now process and understand handwritten equations. Recognizing handwritten equations remains challenging task because handwriting varies significantly across individuals. This variation, combined with the complexity of symbols and noise in scanned images, presents a significant challenge to accurate recognition. Recent advances in image processing and deep learning have enabled effective addressing of this issue.

Although there are many tools available to solve mathematical problems, our study focuses on creating an easy-to-use system with a larger dataset (over 100,000 images) that supports different variables and digits and a PyQt5 interface, making the system ideal for students learning algebra up to degree three equations using Convolutional Neural Network (CNN). Especially during the COVID-19 pandemic and with the rise of work-from-home practices, such a tool can be particularly convenient in digital learning environments, where students can upload their handwritten equations and receive instant solutions. Such works not only save time but also improve

*Corresponding author: Anupa Gaire
Department of Computer Engineering, Khwopa Engineering College, Bhaktapur, Nepal
Email: aanupa.gaire@gmail.com
(Received: September 15, 2025, Accepted: December 31, 2025)
https://doi.org/10.3126/jsce.v13i2.96568

accuracy, making the learning and teaching process more efficient. The system developed here is designed to recognize handwritten digits and mathematical symbols, as well as solve polynomial equations up to degree three (linear, quadratic, and cubic), which are commonly encountered in both academic and real-world applications.

Over the past few decades, significant work has been done in the polynomial expression recognition, with researchers advancing algorithms and applications. These investigations have laid a strong foundation for further exploration. Shuvo et al. (2020) developed a CNN-based handwritten polynomial solver capable of handling quadratic, cubic, and higher-order polynomials. This solver takes an image of a handwritten polynomial, processes it, and provides a simplified solution. Patil and Varma (2022) proposed a CNN-based handwritten expression solver integrated with the SymPy module in Python, which was then automatically solved. The method involves two phases: expression recognition and expression evaluation. Dsouza and Mascarenhas (2018) focus on the offline recognition of mathematical expressions from scanned paper documents. Using different classifiers, using a custom CNN architecture (SpNet-CNN) on 83 symbol classes over 100 epochs. Yadav et al. (2025) utilized a fine-tuned InceptionV3 model for image classification and segmentation, trained on a large dataset of 66,000 images of digits, symbols, and characters, achieving over 94% accuracy on degree-4 polynomial equations. Zhang et al. (2019) developed a CNN–RNN–based end-to-end model for mathematical expression recognition, overcoming OCR limitations with complex symbols and layouts, achieving a Bilingual Evaluation Understudy (BLEU) score of 88.42% on the IM2LATEX-100K dataset. Lu and Mohan (2015) developed a system to convert handwritten mathematical expressions to LaTeX using the CROHME dataset. Their approach combines CNNs for symbol recognition and HMMs for contextual relationships, achieving approximately 90% accuracy and outperforming traditional SVMs, although challenges remain with complex layouts and spatial structures.

Building on these advancements, our work addresses key limitations by developing a CNN-based system tailored explicitly for polynomials of degree three, with a focus on educational purposes. We emphasize a user-friendly PyQt5 interface that supports 16 specific symbol classes (digits $0$–$9, +, -, *, =, x, y$), and also implement data augmentation (from $30,000$ to over $100,000$ samples) to improve robustness across various handwriting styles, which helps us achieve a classification accuracy of 98.99%, and we also integrate SymPy for precise solving. Our approach offers a practical tool for students and educators, particularly on online learning platforms.

The key contribution of this study is the development of a lightweight CNN-based system specifically designed to recognize and solve handwritten polynomial equations of degree up to 3. Unlike prior works that focus only on symbol recognition or require large, complex architectures, our system integrates a custom 16-class CNN, rule-based exponent detection, and symbolic solving through SymPy into a single, easy-to-use educational tool. Another novelty lies in the construction of an expanded dataset of over 100,000 augmented samples tailored for handwritten mathematical expressions, combined with a PyQt5 desktop interface for real-time equation solving. The system's simplicity, efficiency, and educational usability represent a practical innovation compared to existing research on expression recognition. This study is based on research developed during the final year project by (Gaire et al., 2025).

## 2. Methodology

### 2.1. Data Collection and Preprocessing

Datasets for this study were collected from various sources, including publicly available datasets Nano (2017) from the Kaggle platform, academic datasets, and manually collected handwritten digits. We used 16 classes of data (digits $0$–$9, +, -, \times, =, x, y$). The datasets were split into a training and validation set at a 9:1 ratio. This careful data collection and splitting helps ensure that the model can learn effectively and work well when given absolute handwritten equations. First, the raw images were converted to grayscale. Following this inversion, the symbols appeared as white shapes on a dark background, enhancing visibility of the feature and aligning the input format with the CNN's requirements. After inversion, binary thresholding was applied to convert the gray scale images to a black-and-white, effectively removing minor noise and highlighting the boundaries of the symbols. After this process, the data were resized to $28 \times 28$ pixels, a standard dimension widely used in handwritten digit recognition tasks, such as the MNIST datasets of Deng (2012), to ensure uniformity in input size. While the original Kaggle images are $45 \times 45$ pixels, we resized them to $28 \times 28$ to keep the input size consistent across different data sources and to reduce computation. This size is a widely accepted standard in handwritten character recognition, as used in the MNIST dataset, so it offers reliable feature extraction and efficient CNN training. Finally, label encoding was performed to assign each digit and mathematical symbol a unique numerical identifier. For example, digits 0 to 9 are labeled as $0-9$, $+$ as $10$, $'-'$ as $11$, $*$ as $12$, $=$ as 13, x as 14, and y as 15, which served as the target class during model training. This systematic preprocessing pipeline ensured that the dataset was consistent, noise-reduced, and ready for efficient training and classification. To increase

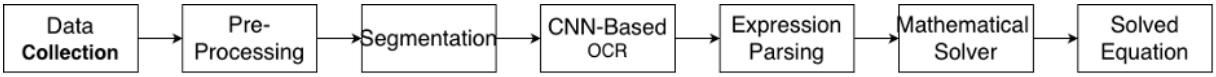


Figure 1. System block diagram

the size and variability of the data set, which is crucial for improving the CNN's generalization capability across various handwriting styles, a custom Python script was applied to grayscale images. Approximately 30,000 raw data points were collected, mainly from Kaggle, and only few 1000 manually handwritten by students and friends who volunteered for this project.

All the data were further expanded using techniques such as ±10° rotations (to simulate natural variations and slight tilts in handwriting) and ±180° rotations(to handle extreme orientation changes, such as inverted or upside-down symbols during scanning or capture). Although it can be rotated by 90° or 270°, we did not include these in our augmentation pipeline because they disrupt the natural layout of mathematical symbols, making digits and operators vertical. Our inputs are expected to be upright or slightly tilted, with ±10° rotations. We included 180° rotations to simulate upside-down scans, while preserving the horizontal symbol layout. Although rotation was the main augmentation used in this study, we also applied several pre-processing steps to handle common noise in handwritten images. Issues such as shadows, uneven lighting, pen-pressure variation, background texture, and minor stray marks were often present. To reduce these effects, we used Gaussian smoothing to reduce speckle noise, erosion and dilation to clean artifacts and strengthen strokes, and contrast normalization to handle uneven illumination before thresholding.

The block diagrams of our proposed system are presented in Figure 1. After data collection, the data were subjected to several additional pre-processing steps using OpenCV to enhance the quality of the input data and improve the model's accuracy before training.

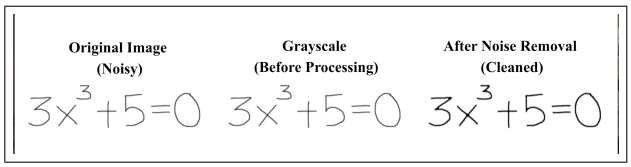


Figure 2. Before/after noise removal on test image

Figure 2 is the original noisy equation (left), gray scale before processing (middle) highlights issues, processed version (right) is cleaner, leading to fewer false detection in segmentation. Figure 3 illustrates this noise removal pipeline applied to a sample handwritten symbol from our datasets ($'x'$), demonstrating the progressive cleaning of common artifacts. These steps improved segmentation and reduced misclassifications.

After collecting and pre-processing all images, the next crucial step is to organize them into a structured format, allowing for easy use in model training. In this stage, each pre-processed image is paired with its corresponding label. To prepare the datasets, all pre-processed images are converted into numerical arrays (pixel values). Each image array is linked to its correct label. Figure 4 presents the sample images from the datasets used to train the model. Both the image data and labels are stored together in a tabular format, making it easy to handle programmatically. After completing the pairing process, the data set is compiled into a single file named `solver_train_data.csv`. This file acts as the primary input for training our CNN model. Having all the data in one place makes it easier to load it into the training script, shuffle it, and split it into training and validation sets without worrying about managing thousands of separate image files.

## 2.2. Model Architecture

The CNN is a type of deep learning network that has been applied to make predictions of various kinds of data, including text and audio (Lu and Mohan, 2015). A lightweight custom CNN was designed to ensure both computational efficiency and reliable symbol recognition. The architecture is described as follows:

- **Convolutional Layers**: Three convolutional layers, each with ReLU activation, were used to extract essential features from the input symbols.
  Conv Layer 1: 32 filters, $3\times3$ kernel, ReLU activation, followed by MaxPooling ($2 \times 2$).
  Conv Layer 2: 64 filters, $3\times3$ kernel, ReLU activation, followed by MaxPooling ($2 \times 2$).
  Conv Layer 3: 64 filters, $3\times3$ kernel, ReLU activation, followed by MaxPooling ($2 \times 2$).
- **Pooling Layers:** MaxPooling layers with a pool size of $2 \times 2$ were applied to reduce spatial dimensions and minimize overfitting.
- **Flattening and Dense Layers**: The final feature maps were flattened into a 1D vector and passed through two dense layers: one hidden layer with ReLU activation and an output layer with Softmax activation corresponding to the symbol classes.

### 2.2.1 Model Design and Training

After preprocessing and cleaning the handwritten symbol, we built a custom CNN using the Keras library. The dataset

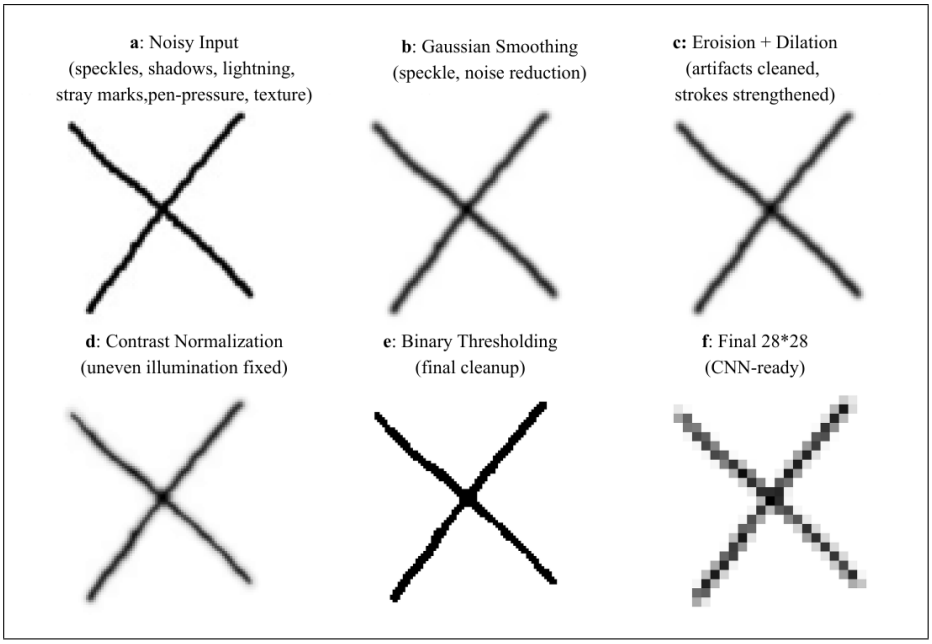


Figure 3. Noise removal pipeline for handwritten symbols

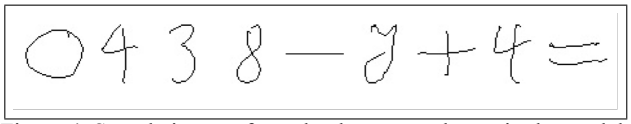

Figure 4. Sample images from the dateset used to train the model

is saved in the file `solver_train_data.csv`, which is loaded into the program first. Each image in this dataset consists of pixel values ranging from 0 to 255. To make the data easier for the CNN to process, these pixel values are normalized by dividing each by 255, which scales them to range 0 to 1. This normalization enables the model to learn more effectively by maintaining consistent input values. Next, the image data is reshaped into a four-dimensional array with the shape (samples, 28, 28, 1). Here, 'samples' represents the number of images; 28 and 28 denote the height and width of each image in pixels respectively; and 1 indicates that the images are grayscale with only one-color channel. Finally, the labels for each image are converted to one-hot-encoded vectors using the Keras categorical() function. One-hot encoding means that each label is represented as a vector with only the index of the correct class set to 1, and all other indices set to 0. This format is required for multiclass classification tasks, enabling the CNN to output the probability of each class for a given input image. In our case, these categories include:

- **Digits:** The numerical symbols from 0 to 9.
- **Mathematical Symbols:** Common operators such as plus $(+)$, minus $(-)$, multiplication $(\times)$, and equal $(=)$.
- **Characters:** Alphabetic variables such as $x$ and $y$, which frequently appear in mathematical expressions.

#### 2.2.2 Model Compilation and Training

The complete model was trained for 40 epochs with a batch size of 64 using the RMSprop optimizer and categorical cross-entropy loss, with a 10% validation split. The custom CNN architecture is trained with Keras, focusing on high-accuracy digit and symbol recognition. The model is trained on the preprocessed dataset (X_train, Y_train).

- **X train:** Contains the input image data, each image is a 28×28 grayscale matrix, normalized and reshaped to (28, 28, 1).
- **Y train:** Contains the corresponding labels for the images, converted into one-hot encoded vectors for multi class classification.

The training was conducted on machines running macOS, equipped with an 8-core CPU and 16 GB of RAM.

#### 2.2.3 Model Saving

After the training process is completed, the model architecture is saved in a JSON file named solver $model.json$, which contains the detailed structure of the neural network, including the layers and their configurations. The trained weights, which represent the learned parameters of the model, are saved separately in an HDF5 file (Fortner, 1998) named $solver_model_weights.h5$.

Table 1. CNN model architecture summary

| Layer (type) | Output Shape | Param # |
|---|---|---|
| conv2d (Conv2D) | (None, 26, 26, 32) | 320 |
| max_pooling2d (MaxPooling2D) | (None, 13, 13, 32) | 0 |
| conv2d_1 (Conv2D) | (None, 11, 11, 64) | 18,496 |
| max_pooling2d_1 (MaxPooling2D) | (None, 5, 5, 64) | 0 |
| conv2d_2 (Conv2D) | (None, 3, 3, 64) | 36,928 |
| flatten (Flatten) | (None, 576) | 0 |
| dense (Dense) | (None, 64) | 36,928 |
| dense_1 (Dense) | (None, 16) | 1,040 |
| | **Total params** | 93,712 (366.06 KB) |
| | **Trainable params** | 93,712 (366.06 KB) |
| | **Non-trainable params** | 0 (0.00 Byte) |

Table 1 presents the model summary and the parameters used during training, with the model defined as a Sequential Model. The model consists of convolutional, max pooling, flattening, and dense layers, totaling 93,712 trainable parameters. The parameter counts in are calculated using the standard formulas for convolutional and dense layers. For each Conv2D layer, the parameters are calculated as: (kernel height × kernel width × input channels × number of filters) + number of filters.

For example, the first Conv2D layer uses 32 filters of size 3×3 with 1 input channel, giving (3×3×1×32) + 32 = 320 parameters.

For Dense layers, the calculation is: (input units × output units) + output units. These formulas are applied to all layers, and the values are summed to get the final total of 93,712 trainable parameters in the model.

Its summary was generated using `model.summary()` and saved to `model_summary.txt` for documentation. We did not use complex architectures such as ResNet or Inception because our task involves small, grayscale 28×28 handwritten symbols, where a lightweight model is sufficient. Such deep networks are designed for large, high-resolution images with rich color and texture, which is not the case here. A simple Sequential CNN offers lower computational cost, avoids overfitting on a small dataset, trains faster, and is easier to deploy and understand.

## 2.3. Model Evaluation

The model's performance is evaluated using various metrics to assess its accuracy and generalization.

- **Character Recognition Accuracy:** Correct identification of digits, operators, and variables.
- **Equation Parsing Accuracy:** Evaluating how well the model recognizes and structures the equation.
- **Solution Verification:** Ensure that the calculated solution matches the expected result for test cases.

## 2.4. Model Testing and Post-Preprocessing

After the training phase was complete, the model was evaluated on a separate test set that had not been used during training. This testing data helps us to verify how well the model performs on new, unseen handwritten equations.

### 2.4.1 Image Segmentation and Symbol Recognition

Image segmentation and symbol recognition are performed to extract individual handwritten symbols from an input equation image and to classify each symbol using the trained CNN model. Here, it segments the image into separate lines using horizontal contour detection, then converts the image to grayscale and applies adaptive thresholding. It uses morphological operations to group line contours, extracts individual line images from the input, and returns them. Again, it takes each line and further segments it into individual symbols, applying grayscale conversion, thresholding, and morphological operations to detect symbol contours, extract bounding boxes, and resize each symbol to 28 × 28 pixels. And then returns a list of segmented and cleaned symbol images, ready for classification. This segmentation ensures that each symbol is accurately isolated, cleaned, and formatted for the CNN model's input.

The segmentation stage then extracts individual symbols, which are recognized by a CNN-based Optical Character Recognition (OCR) model that identifies digits and variables (e.g., x, y, operators +, -, =, etc.). After segmentation, each symbol undergoes preprocessing to match the format used during training, and the preprocessed symbol is passed through the trained CNN. The model predicts a class label from the 16 symbol categories (digits 0–9 and symbols +, –, *, =, x, y). Finally, the predicted class labels are mapped back to their corresponding symbols and combined in the correct sequence to reconstruct the full mathematical expression.

To handle polynomial exponents (like $x^2$ or $y^3$), which

are not separate symbols but small digits written above variables, we use the position and size of each symbol's bounding box. During contour detection, we check the height (exponents are less than 60% of the base symbol) and vertical position (superscripts are at least 20% above the baseline). Symbols are read left-to-right, and small digits above a variable are grouped with it. This rule-based method allows us to correctly interpret expressions like "$x^2$" without needing a separate '^' symbol. Testing on 150 polynomial images showed that 90% of exponents were correctly assigned, achieving an overall classification accuracy of 98.99%.

#### 2.4.2 Equation Parsing and Structuring

Once the characters and symbols are recognized, the next step is to parse the equation. The recognized characters are arranged in the correct order to form a mathematical expression, and the equation is structured as a sequence to capture the relationships between operators and operands, which is crucial for accurate mathematical interpretation. For powers, after segmentation, we use position rules: digits right after 'x' or 'y' are checked for superscript traits (smaller height and higher position) and treated as exponents. For example, "$x^2 =$" is interpreted as "$x^2 = 0$" for SymPy. This method relies on typical handwriting, where exponents are small and raised, allowing us to handle polynomials up to degree 3 without additional symbol classes.

#### 2.4.3 Equation Solving

Once the mathematical expression is structured, it is passed to a Solver Integration. The recognized equation is parsed using custom logic to identify variables. Based on the structure, the equation is evaluated using either Python's built-in functions for simple arithmetic expressions or symbolically solved using the SymPy library for other polynomial equations. For arithmetic expressions without variables, the solution will be calculated using Python's eval() function. For algebraic equations with variables, SymPy will be used to symbolically solve them. The computed solution is then converted into a readable string format and displayed to the user.

### 2.5. User Interface

The UI was built using PyQt5, a comprehensive Python binding for Qt v5.Qt is a popular set of cross-platform C++ libraries for developing modern desktop applications **Riverbank2024**. It was designed to be simple, allowing users to upload handwritten equation images via drag-and-drop or file selection, choose the equation type, and have the interface adapt accordingly. Key buttons include opening images, clearing the display, solving the equation, and viewing the solution history. The recognized equations and their results are displayed in a text area. This interface serves as the main interaction point for using the trained model during inference. The UI includes a differentiation button as a placeholder for future features, but this function is not yet implemented. In the current version, only polynomial equation solving works. We plan to add differentiation in future updates by integrating SymPy's symbolic differentiation.

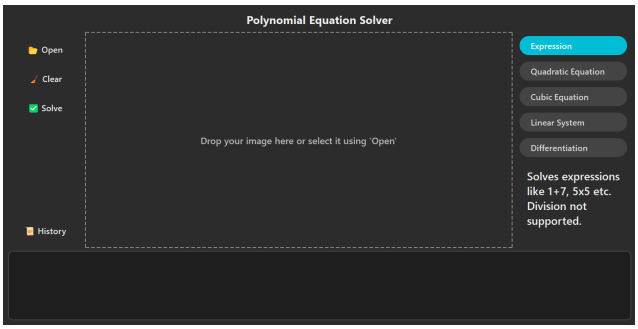


Figure 5. System interface of polynomial equation solver

Figure 5 illustrates the system's primary interface, designed with PyQt5. The interface allows users to open a folder, drag and drop an image, clear the uploaded photo, solve equations, view the solution history, and choose which equation to solve.

This study utilize the Keras library to create and train a CNN that recognize 10 handwritten mathematical symbols. We build the CNN using Keras's Sequential Application Programming Interface (API), which allows us to stack layers to form the network structure. We train the model using Keras's training function over multiple epochs, adjusting the weights to minimize errors. After training, we save the model. This process saves the model's structure, weights, and settings, making it easy to load and reuse the model later without retraining.

## 3. Results and Discussion

The processed images of mathematical symbols were trained using a custom CNN. The photos were trained in 40 epochs. Then, the results obtained from the trained model are evaluated based on their performance, accuracy, and loss. Figure 6 illustrates the training (blue line) and the validation (orange line), the accuracies over 40 epochs. Both metrics show rapid improvement during the initial epochs, where accuracy peaks at $\sim$ 1.0, indicating effective learning. The training accuracy remains stable at this high level, while the validation accuracy shows minor fluctuations in later epochs. Similarly, we present the loss of our model during both the training and validation phases.

Figure 7 shows the training loss represented by the blue line and the validation loss represented by the orange

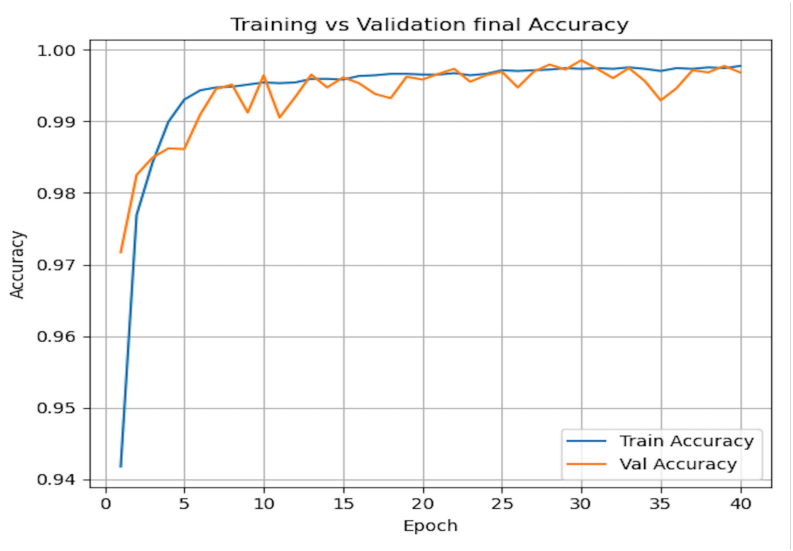


Figure 6. Training vs validation accuracy

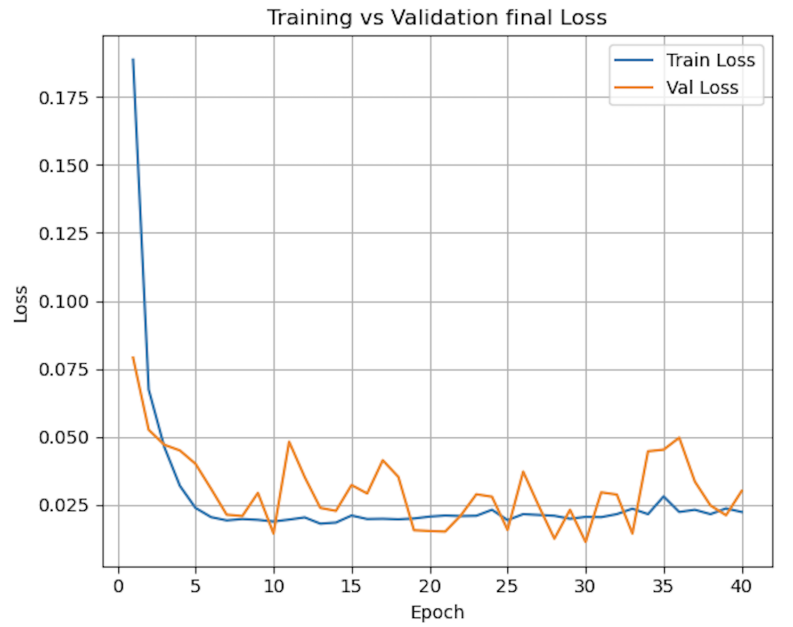


Figure 7. Training vs validation loss

line. Both losses drop rapidly at the beginning, indicating that the model is learning effectively. The training loss becomes very low and stable after approximately 10 epochs, indicating that the model effectively fits the training data. However, the validation loss stops improving and begins to fluctuate slightly after epoch 30, a sign that the model may be beginning to overfit. Therefore, we stop the training, as the best performance was observed around 40 epochs.

After training the model, the equations were passed into the user interface for solving, and the resulting output is shown below: Figure 8 illustrates the system that solves a linear equation and displays the results. If the equation has a solution, the system generates it; otherwise, it shows 'No solution', as not every equation has a valid solution. Figure 9 shows the system successfully solving a quadratic equation, demonstrating its ability to handle more complex mathematical expressions and display accurate results.

Figure 10 shows a history of equations solved through both image uploads in the user interface. This feature enables users to review previously solved problems for reference and verification purposes. Figure 11 illustrates the error message that appears when an incorrect or unsupported

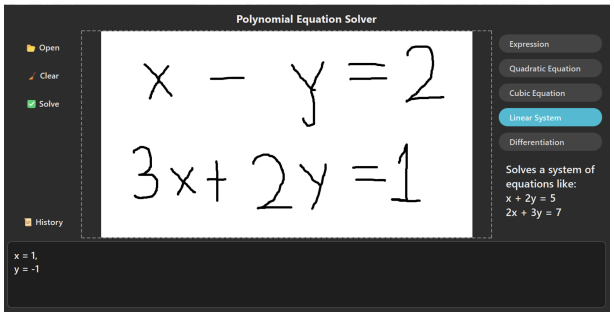


Figure 8. Solving a system of linear equations

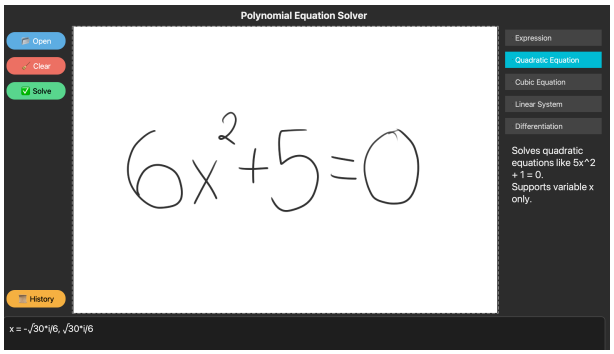


Figure 9. Solving a system of quadratic equation

image is provided as input, highlighting the system's ability to handle invalid inputs.

The system used in this study relies on common handwriting traits, such as smaller digits placed after variables, to detect powers like $x^2$ using bounding-box analysis during segmentation. This allowed the correct parsing of quadratic and cubic equations. There are still a few challenges remain with unusual notations, such as inline fractions or multi-digit exponents, that cause about 10% errors, which are flagged by UI warnings. Our system uses SymPy to solve recognized polynomial equations. Although SymPy can analytically solve quartic (fourth-degree) equations and, in some cases, higher-degree polynomials, our current implementation only supports up to cubic equations. We chose this limit to maintain reliable symbol recognition, ensure efficient equation parsing, and provide a stable user experience suited to typical algebra-level tasks. While adding support for quartic equations is technically possible within our framework, it would require improvements in symbol recognition and interface design. Another limitation is the scope of dataset: It covers polynomials of degree up to 3. Handling higher-degree equations would require more data and training. Future work may extend the system to quartic polynomials using SymPy and an expanded dataset.

Even with 98.99% accuracy, the system can still fail in some cases. The model has difficulty recognizing symbols written with unusual shapes and overlapping strokes. It

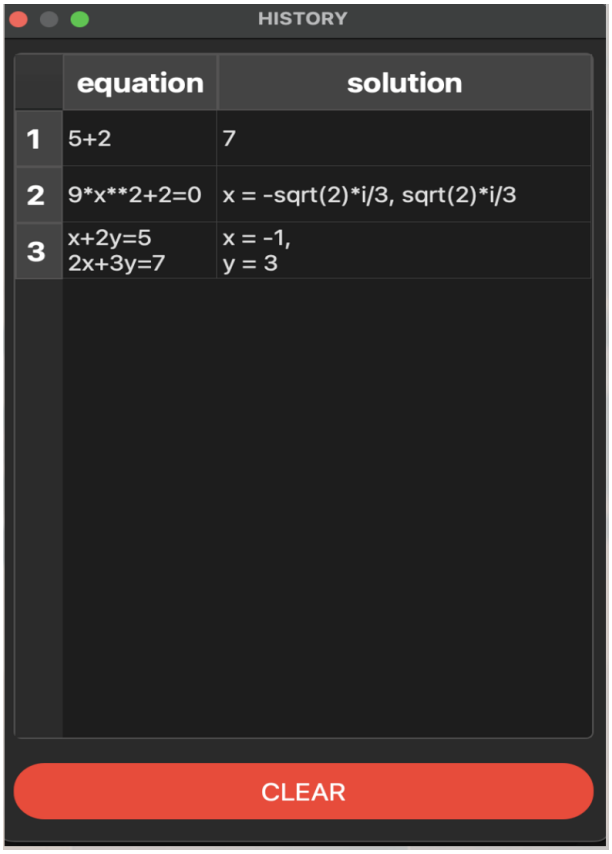


Figure 10. History of solved equations from UI uploads and string input

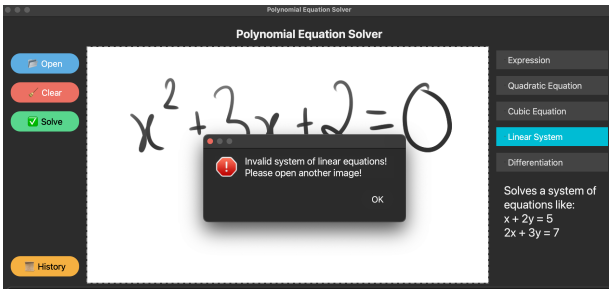


Figure 11. Error shown for incorrect image input

also struggles with complex expressions such as multi-digit exponents and inline fractions, since these patterns are underrepresented in the current dataset. Additional errors occur when input images are poorly scanned, rotated at uncommon angles, or affected by shadows and uneven lighting. To address these issues, future work could expand the dataset with more diverse handwriting samples, add specific classes for superscripts and additional mathematical symbols, integrate an automatic orientation-correction module, and explore transformer-based or attention-driven architectures to better capture spatial relationships between symbols.

Although this study does not include experiments on standard benchmark datasets such as CROHME, we created a custom model tailored to our application. Even without these baselines, we provide a comparative discussion to contextualize our performance within the existing research landscape. Direct symbol-level accuracy comparisons must be interpreted cautiously, as they are influenced by dataset variations in size, handwriting diversity, and noise levels. Nonetheless, our approach shares similarities with prior works in focusing on handwritten mathematical symbols and polynomials. For instance, Shuvo et al. (2020) reported 97–98% accuracy on a CNN-based solver for quadratic/cubic polynomials using 16000 binary image. Yadav et al. (2025) achieved 94% on degree-4 equations with a fine-tuned InceptionV3 model trained on 66,000 images. Patil and Varma (2022) with a CNN-SymPy integration on general expressions. In comparison, our lightweight CNN reaches 98.99% on an expanded, augmented dataset of over 100,000 samples tailored for third-degree polynomials, suggesting competitive performance under more diverse handwriting conditions. Moreover, unlike these methods, our system employs a simpler architecture (93,712 parameters) and integrates rule-based exponent detection with a PyQt5-based end-to-end educational interface, enhancing practical usability. These results demonstrate the efficiency and applicability of our method for educational purposes.

## 4. Conclusion

The study successfully developed a CNN-based system capable of recognizing and solving handwritten polynomial equations of degree up to 3, such as $ax^3+bx^2+cx+d=0$, achieving an impressive training classification accuracy of 98.99%. The system processes handwritten input, identifies symbols, variables, and operators, and computes solutions by integrating the SymPy library into a PyQt5-based user interface designed for educational purposes, helping reduce manual effort and minimize errors. In the future, a canvas can be created, and even higher degrees will be segmented to give better recognition results.

## References


Deng, L. (2012). The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, *29*(6), 141–142.

Dsouza, L., & Mascarenhas, M. (2018). Offline handwritten mathematical expression recognition using convolutional neural network. *Proceedings of 2018 International Conference on Information, Communication, Engineering and Technology (ICICET)*, 1–3. https://doi.org/10.1109/ICICET.2018.8533789

Fortner, B. (1998). The hierarchical data format. *Dr. Dobbs Journal of Software Tools Prof Program*, *23*(5), 42.

Gaire, A., Shrestha, R., Prajapati, R., & Yakami, S. (2025). *Handwritten polynomial equation solver up to degree 3 using CNN* [Final year project, Department of Computer Engineering, Khwopa Engineering College, Purbanchal University, Nepal].

Lu, C., & Mohan, K. (2015). Recognition of online handwritten mathematical expressions using convolutional neural networks [Project report, Standford University].

Nano, X. (2017). *Handwritten math symbols, Kaggle* [Accessed: 2025-09-03]. https://www.kaggle.com/datasets/xainano/handwrittenmathsymbols

Patil, A., & Varma, T. (2022). Handwritten mathematical expression solver using cnn. *International Journal for Research in Applied Science and Engineering Technology*, *10*(9), 1205–1210.

Shuvo, S. N., Hasan, F., Hossain, S. A., & Abujar, S. (2020). Handwritten polynomial equation recognition and simplification using convolutional neural network. *Proceedings of the 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 1–6. https://doi.org/10.1109/ICCCNT49239.2020.9225587

Yadav, P., Shantilal, S. B., Kumar, V., Sihag, P., Sharma, P. K., & Rana, P. (2025). A deep learning approach for recognizing and solving handwritten mathematical equations. *Neural Computing and Applications*, *37*(14), 8759–8772.

Zhang, W., Bai, Z., & Zhu, Y. (2019). An improved approach based onCNN-RNNs for mathematical expression recognition. *Proceedings of the 2019 4th international conference on multimedia systems and signal processing*, 57–61.

*This page is intentionally left blank.*