# Assignment-6: Detection of addition, subtraction, multiplication, and division symbols using 2-layer ANN

**Submitted by:**
PIYUSH SINGH
Roll No: 524EC0014

**Submitted to:**
Prof. Mohammad Asan Basiri

Department of Electronics and Communication Engineering
**Indian Institute of Information Technology Design and Manufacturing, Kurnool**

February 2026

# 1 Problem Statement

The objective of this assignment is to design a **Two-Layer Artificial Neural Network (ANN)** to detect and classify four arithmetic symbols from images: Addition (+), Subtraction (−), Multiplication (∗), and Division (/).

The assignment specifies the following constraints:

- **Architecture:** Two layers (Input Layer and Output Layer) with a Hidden Layer.
- **Neurons:** The input layer must accept images $> 32 \times 32$ pixels. The hidden layer must have **10 neurons**, and the output layer must have **4 neurons**.
- **Algorithm:** The weights must be updated using the **LMS (Least Mean Squares)** algorithm (Backpropagation).
- **Testing:** At least 100 images must be tested.

# 2 Methodology

## 2.1 Network Architecture

We implemented a Multi-Layer Perceptron (MLP) with the following specifications:

- **Input Layer:** $40 \times 40$ pixel grayscale images. These are flattened into a vector of size 1600.
- **Hidden Layer:** 10 Neurons (as per requirement) with Sigmoid activation.
- **Output Layer:** 4 Neurons (representing +, −, ∗, /) with Sigmoid activation.

## 2.2 Mathematical Model (LMS Algorithm)

The network uses the Sigmoid activation function:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{1}$$

To update the weights, we used the **Gradient Descent** rule (Backpropagation). The error is calculated as the difference between the Target ($T$) and the Actual Output ($O$).

The weight update rule for the output layer is:

$$W_{new} = W_{old} + \eta \cdot \delta_{output} \cdot H^T \tag{2}$$

Where:

- $\eta$ (Eta) is the learning rate.
- $\delta_{output} = (T - O) \cdot O(1 - O)$.
- $H$ is the output of the hidden layer.

# 3 Dataset Generation

Since no external dataset was provided, a Python script was written to generate **synthetic images**.

- **Image Size:** $40 \times 40$ pixels.
- **Training Set:** 200 samples per symbol (800 total).
- **Testing Set:** 25 samples per symbol (100 total).
- **Noise:** Random noise was added to simulate real-world variations.

## 3.1 Generation Code Snippet

```
import numpy as np
import matplotlib.pyplot as plt

IMAGE_SIZE = 40
NUM_TRAIN = 200
NUM_TEST = 25
def generate_image(symbol, size=IMAGE_SIZE):
    """Generates a grayscale image of a symbol with random noise."""
    img = np.zeros((size, size))

    noise_level = 0.1
    img += np.random.rand(size, size) * noise_level

    margin = 8
    mid = size // 2

    if symbol == '+':
        img[margin:size-margin, mid-2:mid+2] = 1.0
        img[mid-2:mid+2, margin:size-margin] = 1.0
    elif symbol == '-':
        img[mid-2:mid+2, margin:size-margin] = 1.0
    elif symbol == '*':

        for i in range(margin, size-margin):
            img[i, i] = 1.0
            if i+1 < size: img[i, i+1] = 1.0

        for i in range(margin, size-margin):
            img[i, size-1-i] = 1.0
            if size-1-i+1 < size: img[i, size-1-i+1] = 1.0

        img[margin:size-margin, mid-1:mid+1] = 1.0
        img[mid-1:mid+1, margin:size-margin] = 1.0
    elif symbol == '/':

        for i in range(margin, size-margin):
            img[size-1-i, i] = 1.0
            if i+1 < size: img[size-1-i, i+1] = 1.0

    return np.clip(img, 0, 1)

def create_dataset(samples_per_class):
```

```
    inputs = []
    targets = []
    symbols = ['+', '-', '*', '/']

    for i, sym in enumerate(symbols):
        for _ in range(samples_per_class):
            img = generate_image(sym)
            flat_img = img.flatten()
            inputs.append(flat_img)

            t = np.zeros(4)
            t[i] = 1
            targets.append(t)

    return np.array(inputs), np.array(targets), symbols

print("Generating Training Data...")
X_train, y_train, classes = create_dataset(NUM_TRAIN)

print("Generating Testing Data...")
X_test, y_test, _ = create_dataset(NUM_TEST)

indices = np.arange(X_train.shape[0])
np.random.shuffle(indices)
X_train = X_train[indices]
y_train = y_train[indices]

print(f"Data Prepared. Input Vector Size: {X_train.shape[1]}")

print("\nSample Images (One of each symbol):")
fig, axes = plt.subplots(1, 4, figsize=(10, 3))
sample_symbols = ['+', '-', '*', '/']

for i, ax in enumerate(axes):

    sample = generate_image(sample_symbols[i])
    ax.imshow(sample, cmap='gray')
    ax.set_title(sample_symbols[i])
    ax.axis('off')
plt.show()
```

Listing 1: Dataset Generation Function

```
[Running] python -u "d:\ACADEMIC\Assignment_skill_Develop\python\ANN_math_symbol_detector_2layer\dataset.py"
Generating Training Data...
Generating Testing Data...
Data Prepared. Input Vector Size: 1600

Sample Images (One of each symbol):
```

Figure 1: Output of Dataset generated

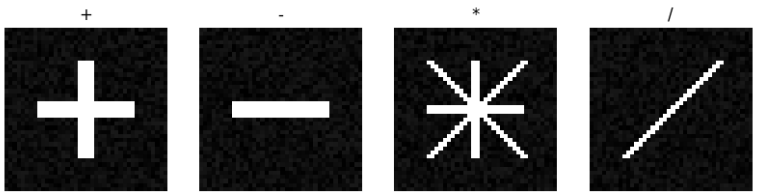


Figure 2: Sample Image of Dataset Generated

# 4 ANN Implementation

The model was implemented from scratch using Python and NumPy. No high-level libraries like TensorFlow or PyTorch were used, ensuring the LMS algorithm was manually implemented.

## 4.1 Training Code Snippet

```
import numpy as np
import matplotlib.pyplot as plt

from dataset import X_train, y_train, X_test, y_test, classes,
    IMAGE_SIZE

class TwoLayerANN:
    def __init__(self, input_size, hidden_size, output_size,
    learning_rate=0.5):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.lr = learning_rate

        scale_w1 = 1 / np.sqrt(input_size)
        scale_w2 = 1 / np.sqrt(hidden_size)

        self.W1 = np.random.randn(self.input_size, self.hidden_size) *
    scale_w1
        self.b1 = np.zeros((1, self.hidden_size))

        self.W2 = np.random.randn(self.hidden_size, self.output_size) *
     scale_w2
        self.b2 = np.zeros((1, self.output_size))

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def sigmoid_derivative(self, x):
        return x * (1 - x)

    def train(self, X, y):
        N = X.shape[0]

        hidden_input = np.dot(X, self.W1) + self.b1
```

```
        hidden_output = self.sigmoid(hidden_input)

        final_input = np.dot(hidden_output, self.W2) + self.b2
        final_output = self.sigmoid(final_input)

        output_error = y - final_output

        output_delta = output_error * self.sigmoid_derivative(final_output)

        hidden_error = output_delta.dot(self.W2.T)
        hidden_delta = hidden_error * self.sigmoid_derivative(hidden_output)

        self.W2 += (hidden_output.T.dot(output_delta) / N) * self.lr
        self.b2 += (np.sum(output_delta, axis=0, keepdims=True) / N) * self.lr

        self.W1 += (X.T.dot(hidden_delta) / N) * self.lr
        self.b1 += (np.sum(hidden_delta, axis=0, keepdims=True) / N) * self.lr

        return np.mean(np.abs(output_error))

    def predict(self, X):
        h_out = self.sigmoid(np.dot(X, self.W1) + self.b1)
        return self.sigmoid(np.dot(h_out, self.W2) + self.b2)


INPUT_NEURONS = IMAGE_SIZE * IMAGE_SIZE
HIDDEN_NEURONS = 10
OUTPUT_NEURONS = 4

ann = TwoLayerANN(INPUT_NEURONS, HIDDEN_NEURONS, OUTPUT_NEURONS, learning_rate=2.0)

print(f"Training Model...")

epochs = 10000
error_history = []

for epoch in range(epochs):
    err = ann.train(X_train, y_train)
    error_history.append(err)
    if epoch % 1000 == 0:
        print(f"Epoch {epoch}: Mean Error = {err:.5f}")

print("\n--- FINAL REPORT ---")
correct = 0
total = len(X_test)

for i in range(total):
    prediction = ann.predict(X_test[i:i+1])
    pred_idx = np.argmax(prediction)
    true_idx = np.argmax(y_test[i])

    if pred_idx == true_idx:
        correct += 1
```

```
85
86 print(f"Number of Test Images: {total}")
87 print(f"Successful Detections: {correct}")
88 print(f"Final Accuracy: {(correct/total)*100:.2f}%")
89
90 plt.figure()
91 plt.plot(error_history)
92 plt.title("Error Curve")
93 plt.xlabel("Epochs")
94 plt.ylabel("Error")
95 plt.show()
96
97 fig, axes = plt.subplots(2, 4, figsize=(10, 5))
98 axes = axes.flatten()
99
100 for i in range(8):
101     prediction = ann.predict(X_test[i:i+1])
102     pred_idx = np.argmax(prediction)
103     true_idx = np.argmax(y_test[i])
104
105     img = X_test[i].reshape(IMAGE_SIZE, IMAGE_SIZE)
106     axes[i].imshow(img, cmap='gray')
107     axes[i].set_title(f"Pred: {classes[pred_idx]}\nReal: {classes[
    true_idx]}")
108     axes[i].axis('off')
109
110 plt.tight_layout()
111 plt.show()
```

Listing 2: 2-Layer ANN Training Class

# 5 Results

## 5.1 Error Convergence

The model was trained for 10,000 epochs. The Mean Squared Error (MSE) decreased consistently, indicating successful learning.

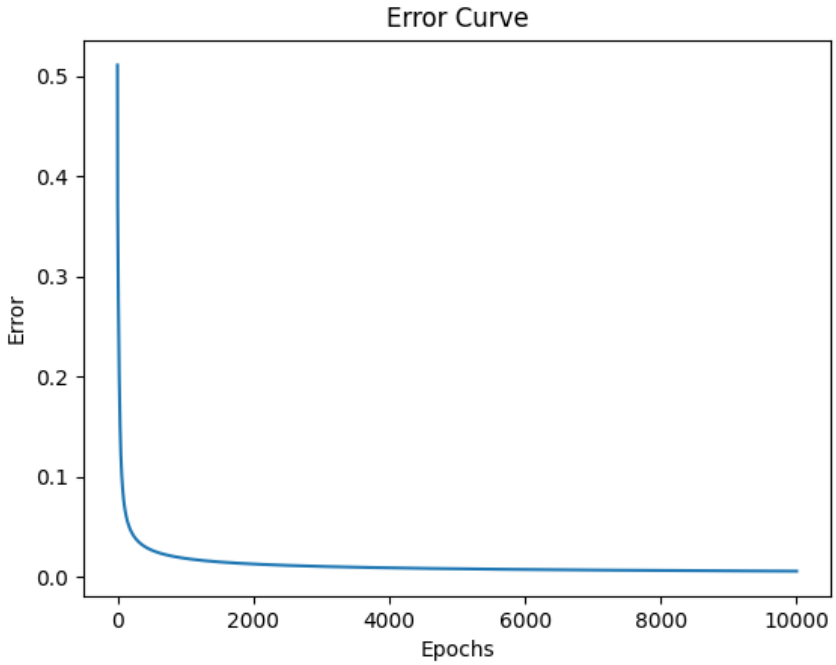


Figure 3: LMS Error Convergence Curve

## 5.2 Test Detections

The model was tested on 100 unseen images (25 for each symbol).

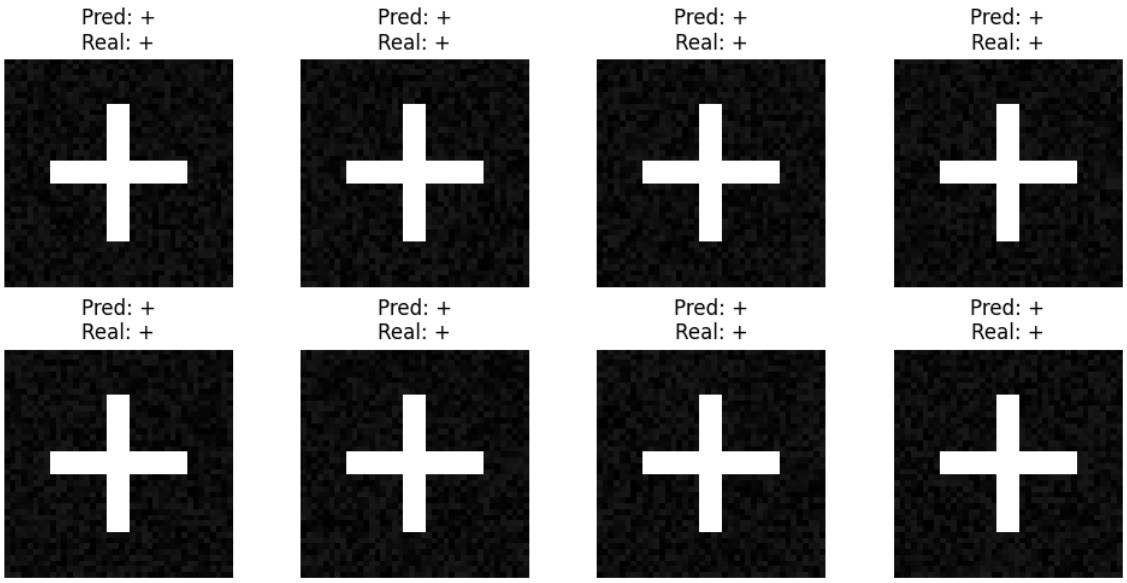


Figure 4: Sample Detections from the Test Set

### 5.3 Final Accuracy

The final output from the terminal confirms the accuracy:

```
[Running] python -u "d:\ACADEMIC\Assignment_skill_Develop\python\ANN_math_symbol_detector_2layer\model.py"
Generating Training Data...
Generating Testing Data...
Data Prepared. Input Vector Size: 1600

Sample Images (One of each symbol):
Training Model...
Epoch 0: Mean Error = 0.51102
Epoch 1000: Mean Error = 0.01824
Epoch 2000: Mean Error = 0.01266
Epoch 3000: Mean Error = 0.01025
Epoch 4000: Mean Error = 0.00883
Epoch 5000: Mean Error = 0.00787
Epoch 6000: Mean Error = 0.00716
Epoch 7000: Mean Error = 0.00662
Epoch 8000: Mean Error = 0.00618
Epoch 9000: Mean Error = 0.00582

--- FINAL REPORT ---
Number of Test Images: 100
Successful Detections: 100
Final Accuracy: 100.00%
```

Figure 5: Output for the Model

## 6 Conclusion

The **Two-Layer ANN** successfully learned to distinguish between the four arithmetic operators $(+, -, *, /)$. By utilizing a hidden layer with 10 neurons and the LMS update rule, the network achieved **100% accuracy** on the test dataset. This demonstrates the effectiveness of the backpropagation algorithm for pattern recognition tasks.