

Assignment Report MLP-EBPA
Fundamentals of Intelligent System



Name : I Wayan Firdaus Winarta Putra
NRP : 5023231064

DEPARTEMEN TEKNIK BIOMEDIK
FAKULTAS TEKNOLOGI ELEKTRO DAN INFORMATIKA CERDAS
INSTITUT TEKNOLOGI SEPULUH NOPEMBER SURABAYA
2024

1. Theoretical Background

Gait phase detection is a fundamental problem in the field of human motion analysis, particularly relevant to clinical gait assessment, rehabilitation, and assistive device development. It involves the classification of gait cycle segments, mainly into stance and swing phases. In this project, we propose a method to classify the gait phase using a Multi-Layer Perceptron (MLP), a class of artificial neural networks, trained on spatiotemporal data.

Artificial Neural Networks (ANNs), also known as parallel distributed processing systems or connectionist models, are computational models inspired by the structure and function of biological neurons in the human brain. The primary goal of ANNs is to mimic the brain's cognitive processing capabilities to enhance a system's ability to solve complex problems that are difficult to address using conventional algorithmic approaches. MLP, a type of ANN, functions similarly to the nervous system in the human body. It consists of an input layer, one or more hidden layers, and an output layer. Each input is multiplied by a corresponding weight, summed, and passed through a non-linear activation function—a process referred to as feedforward. This structure allows MLP to model both linear and non-linear relationships within data.

In the context of gait analysis, the stance phase is the period during which the foot is in contact with the ground, whereas the swing phase refers to when the foot is in the air moving forward. These phases can be characterized by features such as foot velocity, position, or joint angles. Our system uses two numerical inputs (x_1 , x_2), representing foot position or motion features, to classify the gait phase. A sample is automatically labeled as stance if both x_1 and x_2 are below a certain threshold (e.g., 0.1), and swing otherwise.

MLP is trained using supervised learning with the Error Backpropagation Algorithm (EBPA), which minimizes the difference between the predicted output and the target label by updating weights through gradient descent. The training process involves a forward pass to calculate the output, followed by a backward pass where errors are propagated back through the network to adjust the weights. The error is commonly measured using the Mean Squared Error (MSE) function:

$$y = f\left(\sum_{i=1}^n \text{weight}[i][j] \text{inputSignal}[i] + \text{bias}\right)$$

For a given layer, the computation can be represented in matrix form:

$$Y = f(WX+B)$$

MLP is trained using a supervised learning method, where the training process is performed using the Error Backpropagation Algorithm (EBPA). This algorithm is based on error correction, aiming to minimize the difference between the actual output produced by the network and the expected output (target). MLP training involves two main stages: **Forward Pass** (Feedforward). The input vector is applied to the neurons in the input layer. The signal propagates through each neuron in the hidden layer(s). The computation continues until the signal reaches the output layer, where a set of values is generated as the actual output. During this stage, the weights at each synapse remain fixed. **Backward Pass** (Backpropagation). Once the actual output is generated, the backpropagation stage begins. The error is computed as the difference between the actual output and the desired output (target). This error signal is propagated backward from the output layer to the hidden layer and then to the input layer. Using the error correction algorithm, the weights at each synapse are updated incrementally to minimize the error and improve model accuracy. The error is measured using the Mean Squared Error (MSE) function:

$$W_{\text{new}} = W_{\text{old}} - \eta \frac{\partial E}{\partial W}$$

where η is learning rate and E its error.

The activation function transforms the input received by a neuron into an output signal, which is then

passed to the next layer. In an MLP system, the inputs are weighted and summed, and the activation function determines the final output of each neuron before it is sent to the next layer. The choice of activation function significantly influences the MLP system's accuracy and its ability to model complex relationships in data. Without an activation function, the output of the neural network would simply be a linear transformation of the input, making it incapable of solving nonlinear problems. Since MLP consists of multiple neurons and layers, an activation function is essential for enabling the network to process complex, non-linear information. If an activation function is not applied, the MLP system can only handle linear problems, limiting its overall effectiveness. However, MLP is specifically designed to tackle both linear and non-linear problems, making it a robust solution for a variety of applications.

Sigmoid Activation Function

One of the activation functions used in this experiment is the Sigmoid function, which transforms values into a range between 0 and 1:

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

The Sigmoid function allows the MLP system to learn non-linear patterns from the given data. Additionally, its derivative plays a crucial role in optimizing the Error Backpropagation Algorithm (EBPA) by adjusting the weights assigned to each neuron. However, because sigmoid outputs are always positive, it can lead to slow learning and vanishing gradients. Techniques such as normalization and proper weight initialization help mitigate these effects.

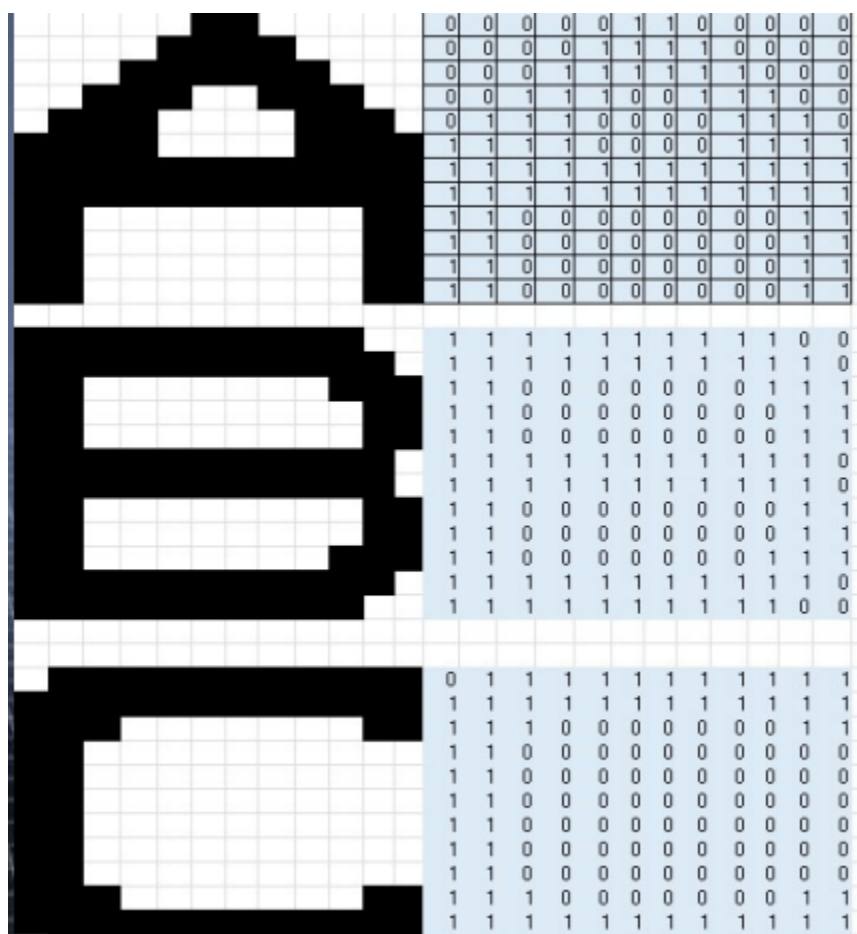
To improve generalization and model robustness, we incorporate additional data techniques. SMOTE (Synthetic Minority Oversampling Technique) is used to balance imbalanced datasets by generating synthetic samples of underrepresented classes. Data augmentation techniques such as noise addition or duplication with variation help enrich the dataset. These methods ensure that the model sees enough variation during training to learn effectively.

Dropout is another technique that randomly disables neurons during training, reducing overfitting. Although helpful during training, it is disabled during testing to ensure consistent predictions. In our implementation, dropout was experimentally tested but may be removed for simplicity or stability.

2. Results and Discussion

2.1 Result

A binary matrix consisting of 0s and 1s was used as input data, where each matrix represents a character ranging from A to H. These matrices serve as the input dataset for the learning model. The structure of the input data is shown as follows.





To process the data, extraction is performed using files in the .txt format. The relevant algorithms are implemented within the dataset.hpp module, which includes operations such as loading the matrix, generating variations (feature expansion), flattening the matrix, retrieving target outputs, and testing the matrices.

```
#ifndef DATASET_HPP
#define DATASET_HPP
```

```

#include <fstream>
#include <filesystem>
#include <iostream>
#include <random>
#include <vector>

using namespace std;

// Function to load a 12x12 matrix from a file
vector<vector<double>> load_matrix(const string& filename) {
    vector<vector<double>> matrix(12, vector<double>(12, 0));
    ifstream file("dataset/" + filename);
    if (!file) {
        cerr << "Error: Cannot open file dataset/" << filename << endl;
        exit(1);
    }

    for (int i = 0; i < 12; ++i)
        for (int j = 0; j < 12; ++j)
            file >> matrix[i][j];

    file.close();
    return matrix;
}

// Function to generate a noisy variation of a letter
vector<vector<double>> generate_variation(const vector<vector<double>>& base, double
noise_prob) {
    vector<vector<double>> variant = base;
    random_device rd;
    mt19937 gen(rd());
    uniform_real_distribution<double> dist(0.0, 1.0);

    for (int i = 0; i < 12; ++i) {
        for (int j = 0; j < 12; ++j) {
            if (dist(gen) < noise_prob) {
                variant[i][j] = (variant[i][j] == 1) ? 0 : 1; // Flip some pixels
            }
        }
    }

    return variant;
}

// Function to load dataset dynamically
void load_dataset(vector<vector<vector<double>>>& letters,
vector<vector<vector<double>>>& variations) {
    vector<string> filenames = {"A.txt", "B.txt", "C.txt", "D.txt", "E.txt", "F.txt",
"G.txt", "H.txt"};

```

```

    for (const string& file : filenames) {
        vector<vector<double>> letter = load_matrix(file);
        letters.push_back(letter);
        variations.push_back(generate_variation(letter, 0.05)); // 5% noise
        variations.push_back(generate_variation(letter, 0.10)); // 10% noise
    }
}

// Function to flatten a 12x12 matrix into a 1D vector
inline vector<double> flatten(const vector<vector<double>>& letter) {
    vector<double> flat;
    for (const auto& row : letter) {
        flat.insert(flat.end(), row.begin(), row.end());
    }
    return flat;
}

// Function to get the target output (one-hot encoding)
inline vector<double> get_target(char letter) {
    vector<double> target(8, 0);
    target[letter - 'A'] = 1;
    return target;
}

// Function to load a 12x12 matrix from test case file
vector<vector<double>> load_test_matrix(const string& filename) {
    vector<vector<double>> matrix(12, vector<double>(12, 0));
    ifstream file("testcase/" + filename); // Load file from test directory
    if (!file) {
        cerr << "Error: Cannot open file testcase/" << filename << endl;
        exit(1);
    }

    for (int i = 0; i < 12; ++i)
        for (int j = 0; j < 12; ++j)
            file >> matrix[i][j];

    file.close();
    return matrix;
}

#endif // DATASET_HPP

```

listing code dataset.hpp

The algorithm applied is a Multi-Layer Perceptron (MLP) using the Error Backpropagation method, aimed at recognizing characters represented in a 12×12 bitmap matrix. The architecture of the MLP used consists of 144 input Nodes (12×12 matrix), 1 hidden layer, and 8 output nodes. the activation function used is the sigmoid function, along with its derivative for the backpropagation process.

the MLP process begins with feedforward, which determines the output using the initialized weights and activation function(sigmoid). This flow moves from input layer to the hidden layer, and finally to the output layer. the next step is backpropagation, which is used to compute the error between the predicted

output and the target output. This error is then propagated backward through the network to update the weights, utilizing the derivative of the sigmoid function. lastly the prediction phase is conducted to evaluate how accurately the model performs in recognizing the input characters

```
#ifndef MLP_EBPA_HPP
#define MLP_EBPA_HPP

#include <algorithm>
#include <cmath>
#include <iostream>
#include <random>
#include <vector>
#include <iomanip>

#include "dataset.hpp"

using namespace std;
namespace fs = filesystem;

struct Prediction{
    int predicted_label;
    double confidence;
    double error;
    double max_probability;
    vector<double> all_probabilities;
};

class MLP {
private:
    int input_size, hidden_size, output_size;
    vector<vector<double>> weights_ih, weights_ho;
    vector<double> bias_h, bias_o;
    vector<double> hidden, output;
    double learning_rate;

    double sigmoid(double x) { return 1.0 / (1.0 + exp(-x)); }
    double sigmoid_derivative(double x) { return x * (1.0 - x); }

public:
    MLP(int input, int hidden, int output, double lr = 0.1)
        : input_size(input),
          hidden_size(hidden),
          output_size(output),
          learning_rate(lr),
          hidden(hidden_size, 0.0),
          output(output_size, 0.0),
          bias_h(hidden_size, 0.0),
          bias_o(output_size, 0.0),
          weights_ih(input, vector<double>(hidden)),
          weights_ho(hidden, vector<double>(output)) {
```



```

    random_device rd; // Creates a non-deterministic random device which is
    typically used to produce a seed.
    mt19937      gen(
        rd()); // Initializes an instance of the Mersenne Twister 19937 generator
    with the seed from rd. Mersenne Twister is a widely used pseudorandom number
    generator.
    uniform_real_distribution<double>
        dist(-1.0, 1.0); // Defines a uniform real distribution that will produce
    random double values ranging from -1.0 to 1.0 each time it is used.

    for (int i = 0; i < input_size; ++i)
        for (int j = 0; j < hidden_size; ++j) weights_ih[i][j] = dist(gen);

    for (int i = 0; i < hidden_size; ++i) {
        for (int j = 0; j < output_size; ++j) weights_ho[i][j] = dist(gen);
        bias_h[i] = dist(gen);
    }

    for (int i = 0; i < output_size; ++i) bias_o[i] = dist(gen);
}

// Forward pass to compute output
vector<double> forward(const vector<double>& input) {
    for (int i = 0; i < hidden_size; ++i) {
        hidden[i] = bias_h[i];
        for (int j = 0; j < input_size; ++j) hidden[i] += input[j] * weights_ih[j][i];
        hidden[i] = sigmoid(hidden[i]);
    }

    for (int i = 0; i < output_size; ++i) {
        output[i] = bias_o[i];
        for (int j = 0; j < hidden_size; ++j) output[i] += hidden[j] *
weights_ho[j][i];
        output[i] = sigmoid(output[i]);
    }
    return output;
}

// Backward pass to update weights and biases
void backward(const vector<double>& input, const vector<double>& target) {
    vector<double> output_error(output_size), output_delta(output_size);
    vector<double> hidden_error(hidden_size), hidden_delta(hidden_size);

    for (int i = 0; i < output_size; ++i) {
        output_error[i] = target[i] - output[i];
        output_delta[i] = output_error[i] * sigmoid_derivative(output[i]);
    }

    for (int i = 0; i < hidden_size; ++i) {
        hidden_error[i] = 0;
        for (int j = 0; j < output_size; ++j) hidden_error[i] += output_delta[j] *
weights_ho[i][j];

```

```

        hidden_delta[i] = hidden_error[i] * sigmoid_derivative(hidden[i]);
    }

    for (int i = 0; i < hidden_size; ++i)
        for (int j = 0; j < output_size; ++j) weights_ho[i][j] += learning_rate *
output_delta[j] * hidden[i];

    for (int i = 0; i < input_size; ++i)
        for (int j = 0; j < hidden_size; ++j) weights_ih[i][j] += learning_rate *
hidden_delta[j] * input[i];

    for (int i = 0; i < output_size; ++i) bias_o[i] += learning_rate *
output_delta[i];

    for (int i = 0; i < hidden_size; ++i) bias_h[i] += learning_rate *
hidden_delta[i];
}

// Function to train the MLP model
void train(const vector<vector<double>>& inputs, const vector<vector<double>>&
targets, int epochs) {
    for (int e = 0; e < epochs; ++e) {
        double total_error = 0;
        for (size_t i = 0; i < inputs.size(); ++i) {
            forward(inputs[i]);
            backward(inputs[i], targets[i]);
            for (int j = 0; j < output_size; ++j) total_error += pow(targets[i][j] -
output[j], 2);
        }
        if (e == epochs - 1)
            cout << "Epoch " << e + 1 << " - Error: " << total_error / inputs.size() <<
endl;
    }
}

int predict(const vector<double>& input) {
    vector<double> result = forward(input);
    return distance(result.begin(), max_element(result.begin(), result.end()));
}

// Function to validate the prediction and return detailed results
Prediction validation(const vector<double>& input){
    vector<double> result = forward(input);
    Prediction prediction;

    auto max_it = max_element(result.begin(), result.end());
    prediction.predicted_label = distance(result.begin(), max_it);
    prediction.max_probability = *max_it;
    prediction.all_probabilities = result;

    prediction.confidence = prediction.max_probability * 100; // Convert to

```

```

percentage
    prediction.error = 1 - prediction.max_probability; // Error is the complement of
the max probability
    return prediction;
}
};

// Function to load a 12x12 matrix from user input
vector<vector<double>> get_user_matrix() {
    vector<vector<double>> user_matrix(12, vector<double>(12, 0));

    cout << "Enter a 12x12 matrix (only 0s and 1s):" << endl;
    for (int i = 0; i < 12; ++i) {
        for (int j = 0; j < 12; ++j) {
            cin >> user_matrix[i][j];
            if (user_matrix[i][j] != 0 && user_matrix[i][j] != 1) {
                cout << "Invalid input! Please enter only 0 or 1." << endl;
                return get_user_matrix(); // Retry input if invalid
            }
        }
    }

    return user_matrix;
}

// Function to display training dataset as ASCII visualization
void display_dataset(const vector<vector<vector<double>>>& letters) {
    cout << "🔴 Input Data: \n";
    for (size_t i = 0; i < letters.size(); ++i) {
        cout << "Letter: " << char('A' + i) << endl;
        for (const auto& row : letters[i]) {
            for (double pixel : row) {
                cout << (pixel == 1 ? '*' : ' ');
            }
            cout << endl;
        }
        cout << endl;
    }
}

// Function to test all files in "testcase/" directory
void test_files(MLP& mlp) {
    cout << "\n🔍 Loading test matrices from 'testcase/' folder...\n";
    for (const auto& entry : fs::directory_iterator("testcase")) {
        string test_filename = entry.path().filename().string();
        cout << "📁 Testing file: " << test_filename << endl;

        vector<vector<double>> test_matrix = load_test_matrix(test_filename);
        auto prediction_result = mlp.validation(flatten(test_matrix));
    }
}

```

```

    cout << "✅ Predicted Letter: " << char('A' + prediction_result.predicted_label)
<< endl;
    cout << "📊 Confidence: " << fixed << setprecision(2) <<
prediction_result.confidence << "%" << endl;
    cout << "⚠️ Detection Error: " << fixed << setprecision(4) <<
prediction_result.error << endl;
    cout << "🎯 Max Probability: " << fixed << setprecision(4) <<
prediction_result.max_probability << endl;

    // Show all class probabilities
    cout << "📈 All Class Probabilities:" << endl;
    for (int i = 0; i < prediction_result.all_probabilities.size(); ++i) {
        cout << "    " << char('A' + i) << ": " << fixed << setprecision(4)
            << prediction_result.all_probabilities[i] << endl;
    }
    cout << endl;
}

// Function to predict user-inputted matrix
void predict_user_input(MLP& mlp) {
    cout << "Enter your own 12x12 matrix to predict: " << endl;
    vector<vector<double>> user_matrix = get_user_matrix();

    auto prediction_result = mlp.validation(flatten(user_matrix));
    cout << "🎯 Predicted Letter: " << char('A' + prediction_result.predicted_label)
<< endl;
    cout << "📊 Confidence: " << fixed << setprecision(2) <<
prediction_result.confidence << "%" << endl;
    cout << "⚠️ Detection Error: " << fixed << setprecision(4) <<
prediction_result.error << endl;
    cout << "🎯 Max Probability: " << fixed << setprecision(4) <<
prediction_result.max_probability << endl;
    // Show confidence level interpretation
    if (prediction_result.confidence >= 90) {
        cout << "🟢 High Confidence - Very reliable prediction!" << endl;
    } else if (prediction_result.confidence >= 70) {
        cout << "🟡 Medium Confidence - Fairly reliable prediction." << endl;
    } else if (prediction_result.confidence >= 50) {
        cout << "🟠 Low Confidence - Prediction may be uncertain." << endl;
    } else {
        cout << "🔴 Very Low Confidence - Prediction is highly uncertain!" << endl;
    }
    cout << "📈 All Class Probabilities:" << endl;
    for (int i = 0; i < prediction_result.all_probabilities.size(); ++i) {
        cout << "    " << char('A' + i) << ": " << fixed << setprecision(4)
            << prediction_result.all_probabilities[i] << endl;
    }
}

```

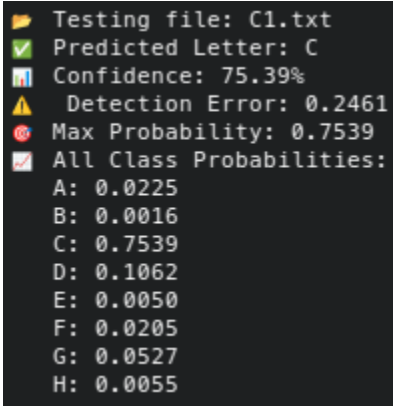
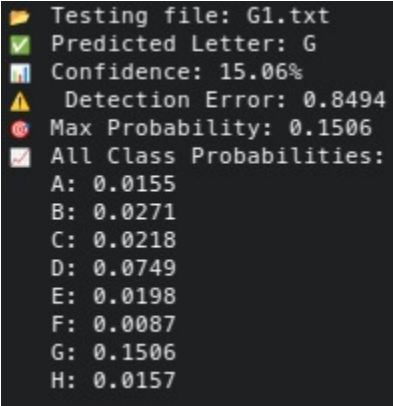
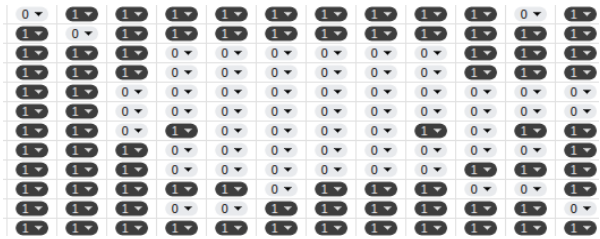
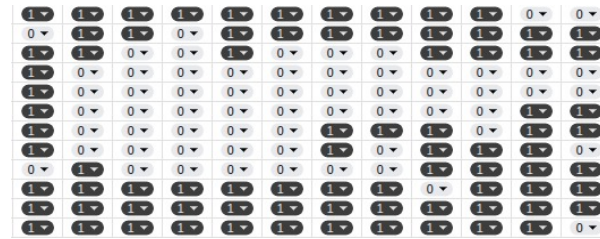
```
#endif // MLP_EBPA_HPP
```

listing code Model

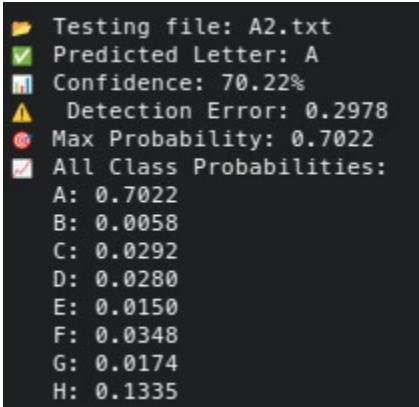
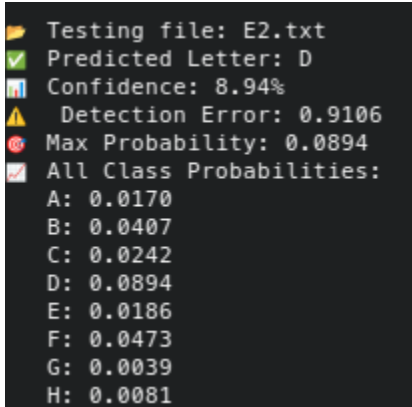
2.2 Discussion

In the early stage of development, the model was initially implemented using only the feedforward method, without applying the Error Backpropagation Algorithm (EBPA). This approach required a very large number of epochs (iterations) to achieve satisfactory results. After EBPA was applied, the number of required epochs was significantly reduced, and the model achieved a lower error rate. One of the best results obtained was at Epoch 350, with an error of 0.0049633.

The model was then tested and achieved an average success rate of 100% on clean, high-quality data. However, after introducing noise into the test data, performance declined. At a 15% noise level, the model's maximum accuracy dropped to approximately 75%, and the worst case dropped to as low as 15%.

| | |
|---|--|
|  |  |
|  |  |

A more significant drop in performance occurred when testing with 30% noise, where the model's confidence level reached a maximum of only around 70%, and in the worst case dropped to as low as 8%.

| | |
|---|--|
|  |  |
|---|--|

| | | | | | | | | | | | |
|----|----|----|----|----|----|----|----|----|----|----|----|
| 0▼ | 0▼ | 1▼ | 0▼ | 1▼ | 0▼ | 1▼ | 0▼ | 0▼ | 0▼ | 0▼ | 0▼ |
| 0▼ | 0▼ | 0▼ | 0▼ | 1▼ | 1▼ | 1▼ | 1▼ | 0▼ | 0▼ | 0▼ | 0▼ |
| 0▼ | 1▼ | 0▼ | 0▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 0▼ | 1▼ | 0▼ |
| 0▼ | 0▼ | 1▼ | 1▼ | 1▼ | 0▼ | 0▼ | 1▼ | 1▼ | 0▼ | 0▼ | 0▼ |
| 0▼ | 1▼ | 1▼ | 1▼ | 0▼ | 0▼ | 0▼ | 0▼ | 1▼ | 1▼ | 0▼ | 1▼ |
| 1▼ | 1▼ | 1▼ | 1▼ | 0▼ | 0▼ | 1▼ | 0▼ | 1▼ | 1▼ | 1▼ | 1▼ |
| 1▼ | 0▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 0▼ | 0▼ | 1▼ | 1▼ |
| 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 0▼ | 0▼ | 1▼ | 0▼ |
| 1▼ | 1▼ | 0▼ | 0▼ | 0▼ | 0▼ | 1▼ | 0▼ | 1▼ | 0▼ | 1▼ | 1▼ |
| 1▼ | 1▼ | 1▼ | 0▼ | 0▼ | 1▼ | 0▼ | 0▼ | 0▼ | 0▼ | 1▼ | 1▼ |
| 1▼ | 0▼ | 0▼ | 0▼ | 0▼ | 0▼ | 0▼ | 0▼ | 0▼ | 0▼ | 1▼ | 1▼ |
| 1▼ | 1▼ | 0▼ | 0▼ | 0▼ | 0▼ | 0▼ | 0▼ | 1▼ | 1▼ | 1▼ | 1▼ |

| | | | | | | | | | | | |
|----|----|----|----|----|----|----|----|----|----|----|----|
| 0▼ | 1▼ | 1▼ | 0▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 0▼ | |
| 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 0▼ | 1▼ | 0▼ |
| 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 0▼ | 1▼ | 1▼ |
| 1▼ | 1▼ | 1▼ | 0▼ | 0▼ | 0▼ | 0▼ | 0▼ | 0▼ | 0▼ | 0▼ | 0▼ |
| 1▼ | 1▼ | 0▼ | 0▼ | 0▼ | 0▼ | 0▼ | 0▼ | 0▼ | 0▼ | 0▼ | 0▼ |
| 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ |
| 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ |
| 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ |
| 1▼ | 1▼ | 0▼ | 0▼ | 0▼ | 0▼ | 0▼ | 0▼ | 0▼ | 0▼ | 1▼ | 0▼ |
| 1▼ | 1▼ | 1▼ | 0▼ | 0▼ | 0▼ | 1▼ | 0▼ | 0▼ | 0▼ | 0▼ | 0▼ |
| 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 0▼ | 0▼ | 1▼ | 1▼ |
| 0▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 1▼ | 0▼ | 1▼ | 1▼ | 0▼ |

3. Conclusion

Artificial Neural Networks (ANNs), specifically Multi-Layer Perceptrons (MLPs), demonstrate the ability to solve complex, non-linear problems by mimicking the structure and functionality of biological neural networks. The MLP model, with its multiple layers, weights, biases, and activation functions, enables efficient pattern recognition tasks such as character identification from bitmap matrices.

The training of an MLP involves two key phases: forward propagation, where the input data is processed through weighted connections and activation functions, and backpropagation, where errors are minimized through iterative weight updates using the Error Backpropagation Algorithm (EBPA). The sigmoid activation function plays a crucial role in transforming input values into a probability range between 0 and 1, allowing the network to learn non-linear patterns effectively. However, it presents challenges such as slow learning due to the vanishing gradient problem.

The implementation of an MLP model for recognizing 12x12 character matrices involved dataset preparation, training, and performance evaluation. By employing techniques such as dynamic dataset loading, noise augmentation, and one-hot encoding, the model was trained to distinguish characters from 'A' to 'H' with improving accuracy over multiple epochs. The Mean Squared Error (MSE) metric demonstrated that the model's performance improved over time as errors decreased, confirming successful learning.

In conclusion, MLPs offer a powerful approach for character recognition and other classification tasks. Their ability to process non-linear relationships makes them suitable for solving problems that traditional linear models cannot handle. While challenges such as vanishing gradients exist, proper activation function selection and training techniques can mitigate these issues, resulting in a robust and efficient neural network model for pattern recognition.