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

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. The human nervous system consists of an intricate network of interconnected neurons, with the brain serving as the central processing unit. Communication between neurons occurs through electrical impulses called action potentials, and information is transmitted at synapses, where neurotransmitters bridge the gap between presynaptic and postsynaptic neurons.

A Multi-Layer Perceptron (MLP) is an artificial neural network (ANN) framework that functions similarly to the nervous system in the human body. MLP consists of multiple layers, including the input layer, hidden layer(s), and output layer. Typically, MLP networks contain multiple neurons in each layer, interconnected to perform computations that generate the desired output. MLP follows a computational approach where each input is multiplied by its corresponding weight, summed, and then processed through an activation function to produce an output—a process commonly referred to as feedforward. Due to its layered architecture, MLP can effectively model complex functions and handle structured data tasks. A key strength of MLP is its ability to process non-linear data, making it suitable for solving problems that cannot be addressed using simple linear models. One example of a non-linear problem is the XOR logic function, which cannot be solved using a single-layer perceptron but can be handled efficiently by MLP. By leveraging its multi-layer structure, MLP can automatically extract features and overcome non-linearity in datasets, making it a powerful tool for complex problem-solving. Mathematically, the output of a neuron can be represented as:

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

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

$$Y = f(\text{WX+B})$$

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:

$$Wnew = Wold - \eta\frac{\partial E}{\partial W}$$

where Ƞ 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:

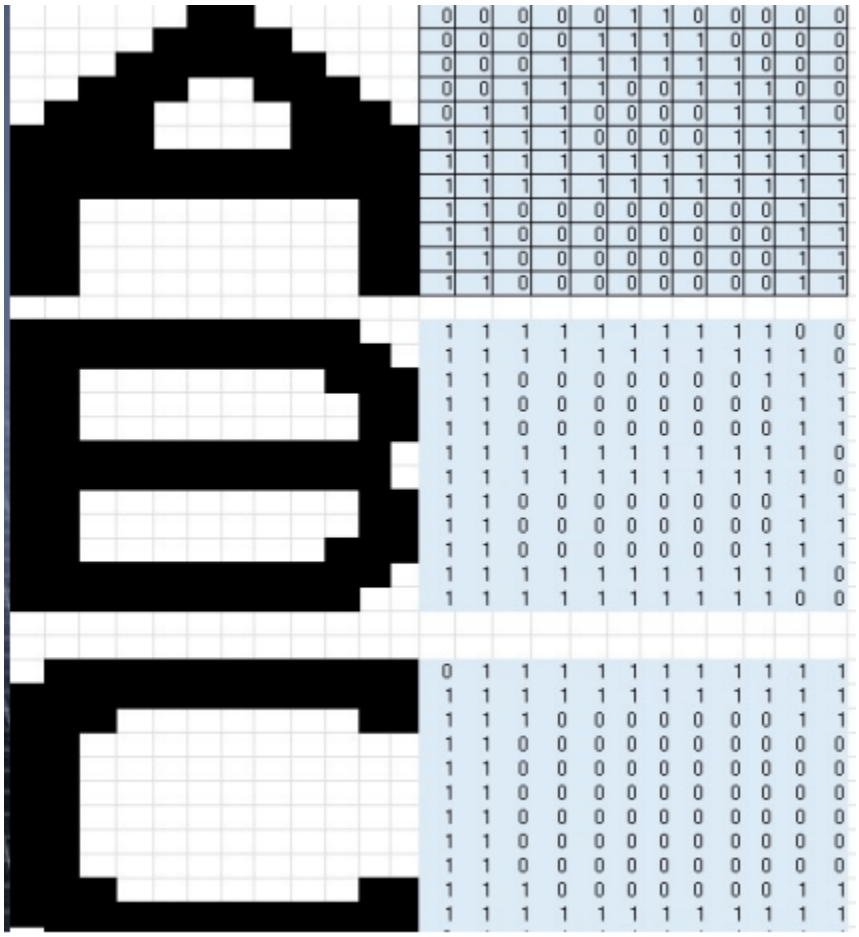
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

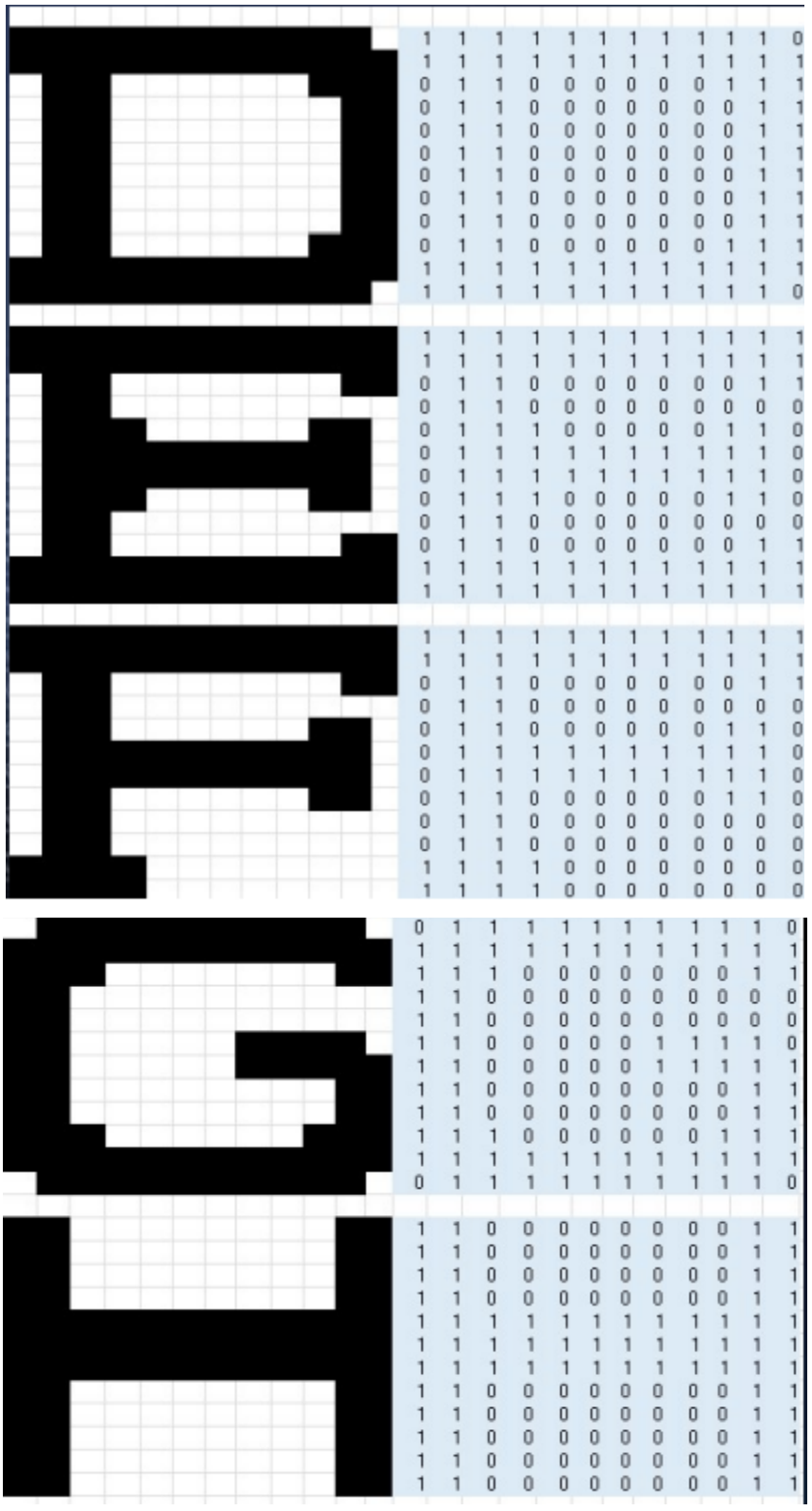
The Sigmoid function allows the MLP system to learn non-linear patterns from the given data. Additionally, its derivative plays a crucial role in optimizing the Error Backpropagation Algorithm (EBPA) by adjusting the weights assigned to each neuron. However, one limitation of the Sigmoid function is that it is not symmetric around zero, meaning that all neuron outputs will have the same sign. This can slow down learning and affect the efficiency of the model.

To address this issue, the Sigmoid function can be scaled to shift its range, making it more balanced around zero. This helps improve weight updates and reduces the likelihood of vanishing gradients, allowing the network to learn more effectively.

2. Results and Data Analysis

The first step is to determine the number of input neurons, hidden layers, outputs, activation function, and training algorithm. Next, create two different types of binary character matrices using Excel and export them as a .txt file. This dataset will include each letter from A to H.





Then, develop an algorithm in C++ that reads the .txt data, converts it into training and test cases, and applies random noise to augment the dataset directly within the dataset.hpp module. The dataset.hpp module contains the following functions:

```cpp
// 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;
}
```

loadMatrixFromFile() – Loads a 12x12 matrix from a file.

```cpp
// 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;
}
```

generateNoisyVariation() – Generates a noisy variation of an alphabet

```cpp
// 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
    }
}
```

loadDatasetDynamically() – Loads the dataset dynamically.

```cpp
// 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;
}
```

flattenMatrix() – Flattens a 12x12 matrix into a 1D vector.

```
// 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;
}
```

getTargetOutput() – Retrieves the target output using one-hot encoding.

```
// 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("test/" + filename); // Load file from test directory
    if (!file) {
        cerr << "Error: Cannot open file test/" << 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;
}
```

loadTestMatrix() – Load a 12x12 matrix from a test case file.

Additionally, the MLP.hpp module is provided, containing all the necessary functions, including the training algorithm, user input handling for the matrix, and test case callbacks. The following program is a basic implementation of a multi-layer perceptron (MLP) using the backpropagation algorithm to recognize character patterns in a 12x12 bitmap matrix.

```
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;
```

variables:
input_size, hidden_size, output_size declared as integer. Define the number of neurons in the input, hidden, and output layers respectively. weight_ih(input to hidden) Weights connecting the input layer to the hidden layer, and weight_ho(hidden to output) Weights connecting the hidden layer to the output layer. bias_h(hidden) Biases for neurons in the hidden layer, bias_o(output) Biases for neurons in the output layer. hidden, output declared as matrix n_x1 Store the active values (or activations) for the neurons in the hidden and output layers during the forward pass. learning_rate is declared as double because this is parameter that controls how much the weights are adjusted during training (backpropagation)

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

activation function

This code implements sigmoid activation function, which transforms input values into range between 0 and 1. This function is non-linearity, which makes the multi-layer perceptron model learn complex patterns from the 12x12 bitmap matrix. by applying the sigmoid function the weight sums from neurons are converted into probabilities, which is crucial for recognizing character from A to H as it normalizes the output.

```cpp
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;
}
```

feedforward function

This function performs a forward pass through the neural network by first computing the hidden layer activations using the input vector, weights, and biases, followed by the application of the sigmoid activation function. It then computes the output layer activations in a similar manner, using the hidden layer activations along with their corresponding weights and biases, followed by another application of the sigmoid activation function.

```cpp
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];
}
```

backpropagation function.

This function calculates the error and corresponding deltas for both the output and hidden layers by comparing the target outputs with the actual outputs computed during the forward pass. It applies the derivative of the sigmoid function to adjust the deltas, then updates the weights connecting the hidden and output layers, as well as the weights connecting the input and hidden layers. Additionally, it updates the bias values for both the hidden and output layers.

```
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);
    }
    cout << "Epoch " << e + 1 << " - Error: " << total_error / inputs.size() << endl;
  }
}
```

training section

In the training phase, the network initializes its weights and biases randomly within a small range. The forward propagation step computes the activations for the hidden and output layers based on the weighted sum of inputs. The backpropagation process then calculates the error by comparing the predicted values with the actual targets. This error is used to update the weights, ensuring that the network improves its predictions over multiple iterations.

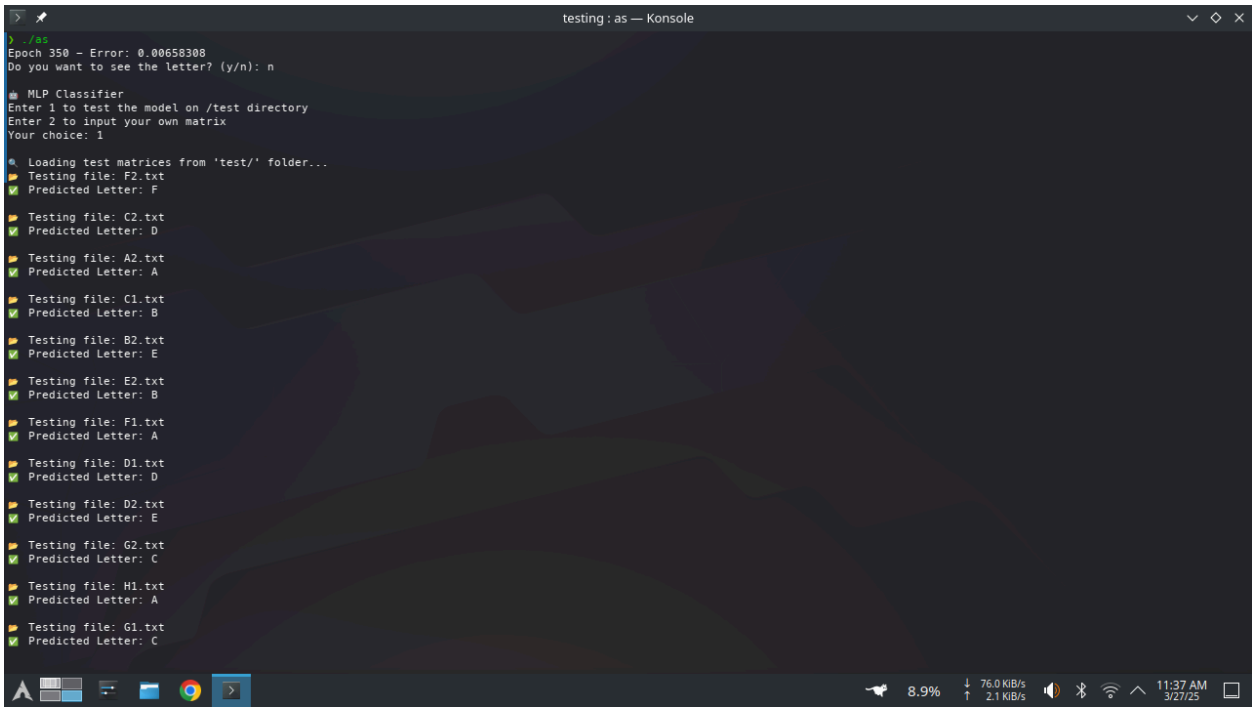
The weight updates occur iteratively across multiple training epochs, gradually reducing the network's error. The model learns by adjusting the weight values in response to the error gradient, ensuring better performance with each epoch. The learning rate determines how much the weights are adjusted during each update.

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

performance analysis

The training process shows a steady reduction in error over time, which indicates that the model is learning effectively. The error is measured using the Mean Squared Error (MSE) formula, which quantifies the difference between the predicted and actual outputs. As the training progresses, the MSE decreases, demonstrating the success of the learning process.

Once trained, the model can predict the character associated with a given 12x12 matrix. The prediction is based on the output neuron with the highest activation value. Testing is performed using matrices stored in the "test/" directory, and the program displays the predicted character for each test input. The results indicate that the MLP successfully recognizes character patterns based on learned representations.



Output using test case

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.