Final Project Report

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
2025

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 (x1, x2), representing foot position or motion features, to classify the gait phase. A sample is automatically labeled as stance if both x1 and x2 are below a certain threshold (e.g., 0.1), and swing otherwise. To reduce the need for manual annotation, this label assignment is performed automatically during the data loading process. Specifically, within the Dataset class, the loadFromFile() function reads x1 and x2 values and assigns a label of 0 (stance) when both are less than 0.1, or 1 (swing) otherwise. This rule is designed to approximate the real-world distinction between stationary and dynamic foot positions and makes the labeling process consistent and efficient across datasets.

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

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

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, 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.

2. Results and Discussion

2.1. Gait-phase result

Given data from the gait of lower limb movement during a complete gait cycle. the classifyPhase function determines the current phase of gait based on sensor readings from the heel and toe, likely using force-sensitive resistors (FSRs). It takes two input values: heel and toe, representing the pressure under the heel and toe of a foot. Depending on the combination of these values, the function returns an integer from 0 to 5, each representing a specific gait phase. If both heel and toe pressures are very low (below 0.03), the function identifies this as the Swing phase (5), indicating the foot is off the ground. When the heel pressure is high (greater than 1.0) and the toe pressure is low, the phase is classified as Initial Contact (0), meaning the heel has just struck the ground. If both the heel and toe show significant pressure (with heel > 1.5 and toe between 0.3 and 0.5), the function returns Foot Flat (1), where the entire foot is contacting the ground. Mid Stance (3) is detected when the heel pressure is moderately high (≥ 0.5) and the toe pressure is also present (> 0.2), suggesting the body is balanced over the foot. When the heel pressure is low (< 0.05) but the toe pressure is high (> 0.44), the phase is Heel Off (2), meaning the heel is lifting off the ground. Toe Off (4) occurs when the heel is almost lifted (< 0.03) and the toe is still in moderate contact (between 0.09 and 0.6), indicating the foot is about to leave the ground. If none of these conditions are met, the function defaults to Swing phase (5), assuming the foot is not in contact with the ground. This logic provides a simple rule-based classification system for gait phase detection in biomechanical or wearable systems. The data remains as follows:

1	0.00E+00	0.00E+00	26	5.00E-03	2.30E-02
2	5.00E-03	1.80E-02	27	1.42E+00	1.80E-02
3	5.00E-03	2.70E-02	28	1.39E+00	2.03E-01
4	1.01E+00	1.80E-02	29	1.87E+00	3.74E-01
5	1.66E+00	3.45E-01	30	1.95E+00	3.94E-01
6	1.87E+00	4.13E-01	31	1.58E+00	3.84E-01
7	1.67E+00	5.69E-01	32	7.08E-01	3.94E-01
8	7.67E-01	8.87E-01	33	2.30E-01	5.50E-01
9	5.00E-03	1.17E+00	34	0.00E+00	7.60E-01
10	5.00E-03	1.37E+00	35	0.00E+00	1.10E+00
11	0.00E+00	1.42E+00	36	5.00E-03	1.26E+00
12	5.00E-03	1.68E+00	37	5.00E-03	1.48E+00
13	5.00E-03	1.66E+00	38	5.00E-03	1.51E+00
14	5.00E-03	1.77E+00	39	5.00E-03	8.53E-01
15	5.00E-03	1.66E+00	40	5.00E-03	9.60E-02
16	5.00E-03	3.99E-01	41	5.00E-03	2.30E-02
17	0.00E+00	4.20E-02	42	5.00E-03	1.80E-02
18	5.00E-03	1.80E-02	43	5.00E-03	1.80E-02
19	5.00E-03	1.80E-02	44	5.00E-03	1.80E-02
20	5.00E-03	1.80E-02	45	5.00E-03	1.80E-02
21	5.00E-03	1.80E-02			
22	5.00E-03	1.80E-02			

23	5.00E-03	2.30E-02	
24	5.00E-03	1.80E-02	
25	5.00E-03	1.30E-02	

The input data used in this model comes from .txt files containing two features, Y1 and Y2. Initially, the signal data was collected without any labels. A separate labeling phase was then carried out by applying specific threshold rules, as described below.

```
int classifyPhase(double heel, double toe) {
  if (heel < 0.03 && toe < 0.03)
    return 5;  // swing
  else if (heel > 1.5 && toe >= 0.3 && toe <= 0.5)
    return 1;  // foot flat
  else if (heel > 1.0 && toe < 0.05)
    return 0;  // initial contact
  else if (heel < 0.05 && toe > 0.44)
    return 2;  // heel off
  else if (heel < 0.03 && toe >= 0.09 && toe <= 0.6)
    return 4;  // toe off
  else if (heel >= 0.5 && toe > 0.2)
    return 3;  // mid stance
  else
    return 5;  // default to swing
}
```

Applied Threshold

These labels serve as the target outputs for training, while the original heel and toe signals (Y1 and Y2) are used as input features to help the model classify the gait phase accurately. the architecture used ia a Multi-Layer Perceptron (MLP) consisting of: 2 input nodes(Y1 and Y2), 1 hidden layer with 8 neurons, and 8 output neurons representing the classes(IC, SW, FF, MSI, HO, TO, Heel, Toe). the model uses the sigmoid activation function and its derivative.

Since the dataset was initially limited in size, data augmentation was applied to improve the model's generalization. The data was augmented 3 times, and each variation was injected with up to ±10% noise. This simulated more realistic and varied input conditions for training.

```
void duplicateAndAugment(int times = 3) {
  std::vector<DataPoint> augmented;
  for (const auto& d : data) {
    for (int i = 0; i < times; ++i) {
      DataPoint aug = d;
      aug.x1 += ((rand() % 2001 - 1000) / 100000.0) * 0.1;
      aug.x2 += ((rand() % 2001 - 1000) / 100000.0) * 0.1;
      augmented.push_back(aug);
    }
  }
  data.insert(data.end(), augmented.begin(), augmented.end());
}
```

implemented Augment-data

During training, the model implements an adaptive learning rate mechanism. If the model's performance does not improve over a certain number of epochs (set to 300 as the patience threshold), the learning rate is reduced. This helps prevent overshooting and stabilizes convergence, particularly in later stages of training.

```
   // Calculate accuracy
    int correct = 0;
    for (size_t i = 0; i < n; ++i) {
      int pred   = predictClass(X[i]);
      int actual = std::distance(Y[i].begin(), std::max_element(Y[i].begin(), Y[i].end()));
      if (pred == actual)
        correct++;
    }

    double avg_loss = total_loss / n;
    double accuracy = 100.0 * correct / n;

    loss_history.push_back(avg_loss);
    accuracy_history.push_back(accuracy);

    // Adaptive learning rate
    if (avg_loss >= prev_loss) {
      no_improve_count++;
      if (no_improve_count >= patience) {
        learning_rate *= lr_decay;
        no_improve_count = 0;
        std::cout << "Reducing learning rate to: " << learning_rate << std::endl;
      }
    } else {
      no_improve_count = 0;
    }

    if (epoch % 10 == 0 || epoch == epochs - 1) {
      double loss_change = prev_loss - avg_loss;
      std::cout << "Epoch " << epoch + 1 << ": Loss = " << avg_loss << ", Accuracy = " << accuracy << "%, LR = " <<
learning_rate << std::endl;

      if (progress_callback) {
        progress_callback(epoch + 1, avg_loss, learning_rate, loss_change);
      }

      prev_loss = avg_loss;
    }

    // Early stopping if we achieve high accuracy
    if (accuracy > 90.0) {
      std::cout << "Early stopping - high accuracy achieved!" << std::endl;
      break;
    }
```

Adaptive learning mechanism

```
FUNCTION sigmoid(x):
    RETURN 1 / (1 + exp(-x))

FUNCTION sigmoid_derivative(x):
    s = sigmoid(x)
    RETURN s * (1 - s)

FUNCTION classifyPhase(heel, toe):
    IF heel < 0.03 AND toe < 0.03:
        RETURN 5  // Swing
    ELSE IF heel > 1.0 AND toe < 0.03:
        RETURN 0  // Initial Contact
    ELSE IF heel > 1.5 AND 0.3 <= toe <= 0.5:
        RETURN 1  // Foot Flat
    ELSE IF heel >= 0.5 AND toe > 0.2:
```

```
      RETURN 3  // Mid Stance
   ELSE IF heel < 0.05 AND toe > 0.44:
      RETURN 2  // Heel Off
   ELSE IF heel < 0.03 AND 0.09 <= toe <= 0.6:
      RETURN 4  // Toe Off
   ELSE:
      RETURN 5  // Default to Swing

// --------------------
// Data Preprocessing
// --------------------

LOAD raw_data.txt  // Contains columns: Y1 (heel), Y2 (toe)
FOR each (Y1, Y2) in raw_data:
   label = classifyPhase(Y1, Y2)
   dataset.append((Y1, Y2, label))

AUGMENT dataset 3x with ±10% noise:
   FOR i in 1 to 3:
      FOR each (Y1, Y2, label) in dataset:
         noisy_Y1 = Y1 * (1 ± random(0, 0.10))
         noisy_Y2 = Y2 * (1 ± random(0, 0.10))
         augmented_dataset.append((noisy_Y1, noisy_Y2, label))

NORMALIZE all Y1, Y2 in dataset to range [0, 1]

ONE-HOT encode labels into 8-class vectors:
   Classes: IC, SW, FF, MSI, HO, TO, Heel, Toe

// --------------------
// MLP Initialization
// --------------------

SET input_size = 2
SET hidden_size = 8
SET output_size = 8
SET learning_rate = 0.01
SET min_learning_rate = 1e-4
SET patience = 300

INITIALIZE:
   - Weights_input_hidden[2][8] with small random values
   - Bias_hidden[8] with small random values
   - Weights_hidden_output[8][8] with small random values
   - Bias_output[8] with small random values

SET best_loss = Infinity
SET no_improvement_epochs = 0

// --------------------
// Training Loop
// --------------------

FOR epoch in 1 to max_epochs:
   total_loss = 0
```

```
   FOR each (Y1, Y2, target_label) in training_dataset:
      // --- Forward Pass ---
      input = [Y1, Y2]

      FOR each hidden neuron h in 1 to 8:
         z_hidden[h] = dot(input, Weights_input_hidden[:, h]) + Bias_hidden[h]
         a_hidden[h] = sigmoid(z_hidden[h])

      FOR each output neuron o in 1 to 8:
         z_output[o] = dot(a_hidden, Weights_hidden_output[:, o]) + Bias_output[o]
         a_output[o] = sigmoid(z_output[o])

      // --- Compute Loss ---
      target = one_hot_encode(target_label)
      sample_loss = sum_over_o((target[o] - a_output[o])^2)
      total_loss += sample_loss

      // --- Backward Pass ---
      FOR each output neuron o:
         error_output[o] = (target[o] - a_output[o]) * sigmoid_derivative(z_output[o])

      FOR each hidden neuron h:
         error_hidden[h] = sum_over_o(error_output[o] * Weights_hidden_output[h][o]) *
sigmoid_derivative(z_hidden[h])

      // --- Update Weights ---
      FOR each hidden neuron h and output neuron o:
         Weights_hidden_output[h][o] += learning_rate * error_output[o] * a_hidden[h]
      FOR each output neuron o:
         Bias_output[o] += learning_rate * error_output[o]

      FOR each input neuron i and hidden neuron h:
         Weights_input_hidden[i][h] += learning_rate * error_hidden[h] * input[i]
      FOR each hidden neuron h:
         Bias_hidden[h] += learning_rate * error_hidden[h]

   // --- Check Adaptive Learning ---
   IF total_loss < best_loss:
      best_loss = total_loss
      no_improvement_epochs = 0
   ELSE:
      no_improvement_epochs += 1

   IF no_improvement_epochs >= patience:
      learning_rate = learning_rate * 0.5
      IF learning_rate < min_learning_rate:
         BREAK  // Early stopping
      no_improvement_epochs = 0

   PRINT "Epoch:", epoch, "Loss:", total_loss, "LR:", learning_rate
```

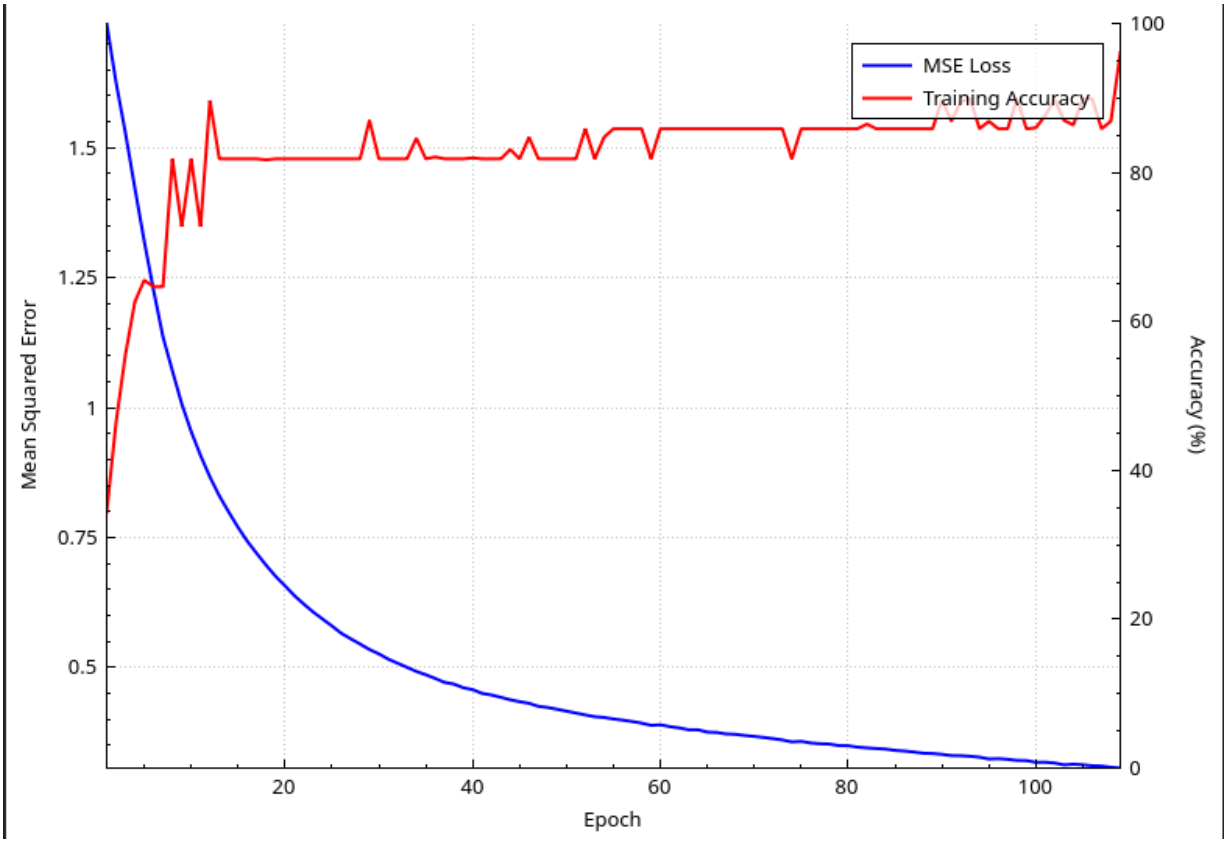
Model Pseudocode

2.2. Gait-Phase discussion

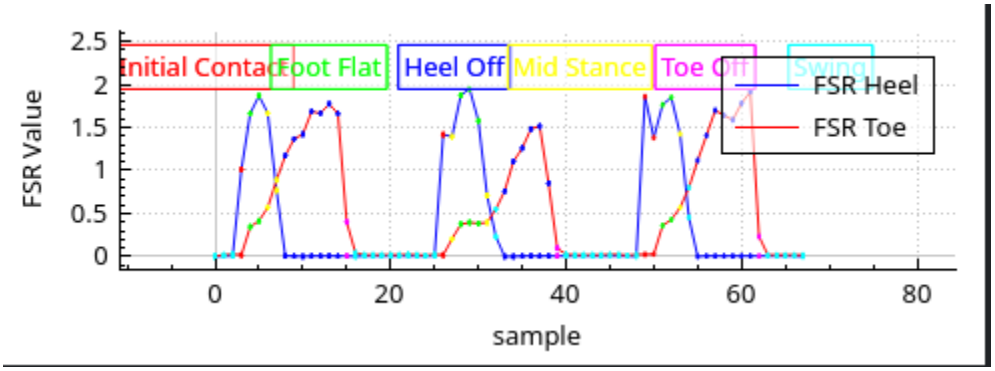
Before the model demonstration, the accuracy remained around 50%, with no significant improvement. This was caused by a logical fallacy during the initial labeling process data that was

expected to be fully labeled turned out to be labeled only with binary values (0 and 1), which oversimplified the classification problem and misrepresented the actual classes.

After applying a proper labeling strategy combined with more effective data augmentation and noise injection the model was able to learn the dataset more efficiently. It achieved a relatively low error (around 1e-2) within fewer than 500 epochs, indicating a significant improvement in training quality.



Train Accuracy: 63.64% (504/792)



Recall Accuracy: 94.12% (128/136)

However, when training is extended to reach an extremely low error threshold (e.g., 1e-10), the model begins to overfit. This overfitting behavior was verified during arrhythmia classification experiments, where the model performed well on the training set but failed to generalize on unseen data.

2.3. Aritmia result

The training dataset consists of 8.4k data points, each with two features: x and y, which represent coordinate points. The goal of the model is to perform clustering using an MLP trained with the Error Backpropagation Algorithm (MLP-EBPA).

The data is provided in .txt format, containing only the x and y values—no labels are included in the input. The corresponding labels are assigned internally during the training process. These labels classify each point into one of the following seven categories(normal, dropped, Tachycardia, R-on-T, PVC, Fussion, Bradycardia). The MLP architecture used is as follows: input layer 2 neurons, 1 hidden layer 3 neurons, and output layer 7 neurons.

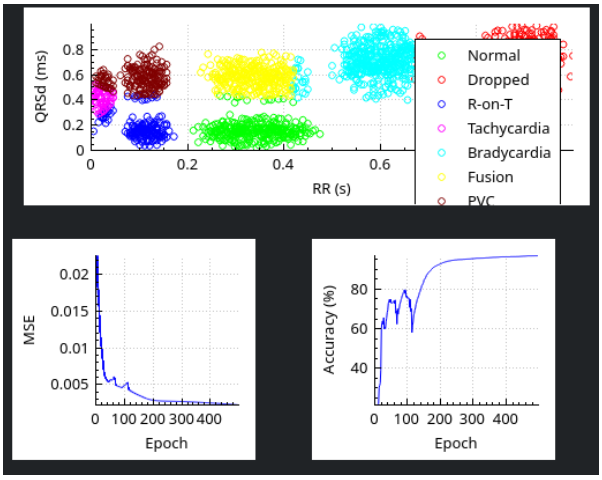
```
Initialize:
   - Input neurons: 2 (x, y)
   - Hidden layer: 6 neurons
   - Output layer: 7 neurons (class labels)
   - Learning rate: η
   - Activation function: sigmoid(x) = 1 / (1 + e^(-x))
   - Derivative of sigmoid: sigmoid'(x) = sigmoid(x) * (1 - sigmoid(x))
   - Weights (random small values):
      - w_input_hidden[2][6]
      - w_hidden_output[6][7]
   - Biases:
      - b_hidden[6]
      - b_output[7]

Load Dataset:
   - Read data from .txt file
   - Each row: (x, y)
   - Labels assigned separately for supervised training

Preprocess:
   - Normalize x and y if necessary
   - One-hot encode the 7-class labels

Training Loop:
   For epoch in 1 to max_epochs:
      For each sample (x, y) with label L:
         // --- Forward Pass ---
         Input = [x, y]

         For each hidden_neuron h in 1 to 6:
            z_h = dot_product(Input, w_input_hidden[:, h]) + b_hidden[h]
            a_h = sigmoid(z_h)

         Hidden_Output = [a_1, a_2, ..., a_6]

         For each output_neuron o in 1 to 7:
            z_o = dot_product(Hidden_Output, w_hidden_output[:, o]) + b_output[o]
            a_o = sigmoid(z_o)

         Predicted_Output = [a_o1, a_o2, ..., a_o7]

         // --- Backward Pass (Error Calculation) ---
         Target = one_hot_encode(L)
         Error_Output = []
         For each output_neuron o:
            error_o = (Target[o] - Predicted_Output[o]) * sigmoid'(z_o)
            Error_Output.append(error_o)
```

```
    Error_Hidden = []
    For each hidden_neuron h:
        error_h = sum_over_o(Error_Output[o] * w_hidden_output[h][o]) * sigmoid'(z_h)
        Error_Hidden.append(error_h)

    // --- Update Weights and Biases ---
    For each hidden_neuron h and output_neuron o:
        w_hidden_output[h][o] += η * Error_Output[o] * a_h
    For each output_neuron o:
        b_output[o] += η * Error_Output[o]

    For each input_neuron i and hidden_neuron h:
        w_input_hidden[i][h] += η * Error_Hidden[h] * Input[i]
    For each hidden_neuron h:
        b_hidden[h] += η * Error_Hidden[h]

// Optional: Evaluate training loss or accuracy
If loss < threshold:
    Break
```

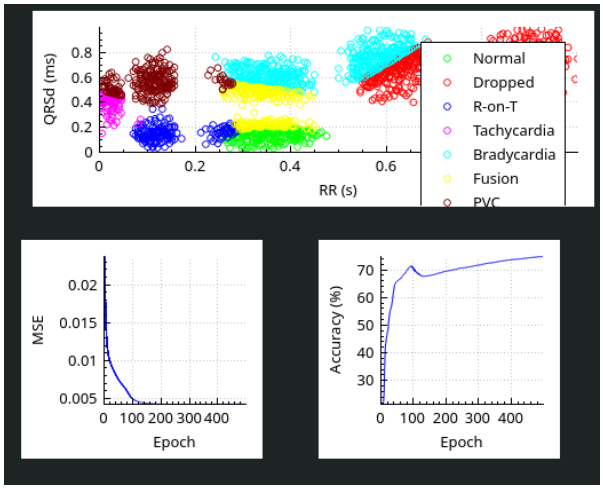
Model Pseudocode

2.4. Aritmia discussion

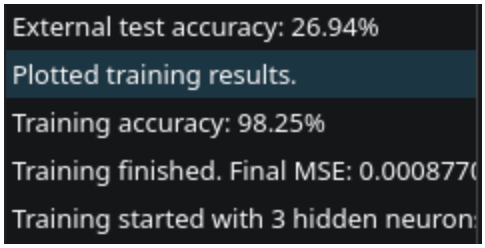
Two experimental setups were conducted to evaluate how the number of neurons in the hidden layer and the choice of MSE thresholds affect model performance. In the first experiment, the number of neurons in the hidden layer was varied to observe changes in classification accuracy. When using 2 neurons, the model achieved over 70% accuracy with an MSE around 1e-2. Notably, the test accuracy was higher than the training accuracy, indicating that the model was more robust and generalized well. Increasing the neuron count to 3 improved training accuracy to around 90%, with a lower error of approximately 1e-4. However, this configuration also introduced signs of overfitting, as the model's performance on test data slightly declined compared to training, highlighting the trade-off between model complexity and generalization.



3 neurons



2 neurons

the second experiment, the effect of a strict MSE target (1e-6 or lower) was tested. While the model was capable of reaching such a small error, doing so led to overshooting during training and excessive computation without significant improvements in real-world performance. In fact, achieving this low MSE threshold caused the model to overfit severely, effectively memorizing the training data. This shows that while reducing error can improve training metrics, setting the MSE too low can be counterproductive wasting resources and harming generalization. Overall, the findings suggest that smaller networks with moderate error thresholds can produce more balanced and robust results, while larger networks and aggressive error targets should be used cautiously to avoid overfitting.

Conclusion

This study explored the effectiveness of MLP-based models for two distinct classification tasks: gait phase detection and arrhythmia classification, focusing on model architecture, labeling accuracy, data augmentation, and error thresholds. In the gait phase task, initial performance was limited due to poor labeling logic. However, after applying a rule-based labeling method along with data augmentation using noise injection, the model's accuracy significantly improved, achieving low training error with fewer than 500 epochs. The use of adaptive learning rate and early stopping further stabilized training and prevented unnecessary computation.

In the arrhythmia classification task, experiments showed that the number of neurons in the hidden layer had a direct impact on accuracy and overfitting. A smaller model with 2 hidden neurons reached over 70% accuracy and generalized well to test data, indicating robustness. Increasing the neurons to 3 improved training accuracy to over 90% but introduced overfitting. Additionally, using very small MSE targets (e.g., 1e-6 or lower) caused overshooting and wasted resources, offering no real-world benefit and increasing overfitting risks.

Overall, the results confirm that proper labeling, moderate model complexity, and practical error targets are crucial for building efficient and generalizable neural networks. Smaller networks with balanced training goals tend to be more robust, while aggressive optimization without control mechanisms can lead to overfitting and inefficiency.