Assignment Report Gait-Phase-recognition

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 (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:

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

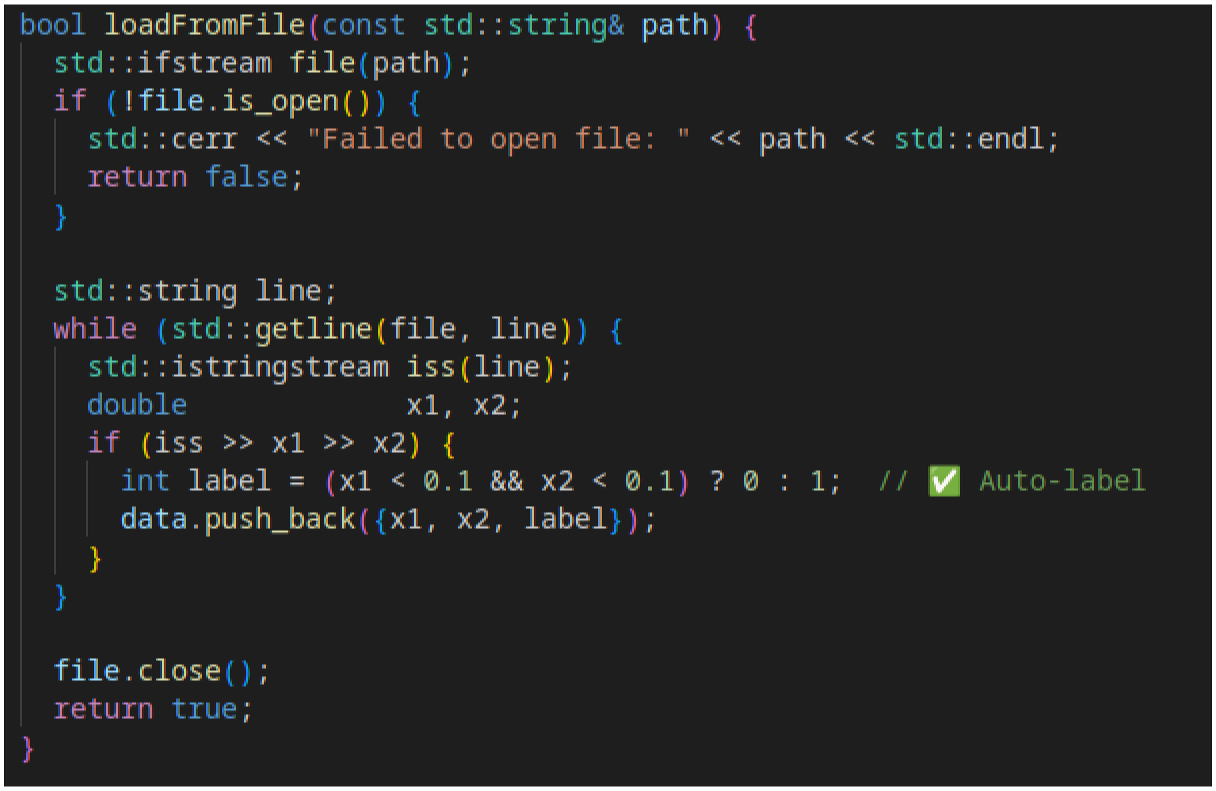
The Sigmoid function allows the MLP system to learn non-linear patterns from the given data. Additionally, its derivative plays a crucial role in optimizing the Error Backpropagation Algorithm (EBPA) by adjusting the weights assigned to each neuron. However, because sigmoid outputs are always positive,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

2. Results and Data Analysis

Given data from the gait of lower limb movement during a complete gait cycle, whenever the Y-axis value is greater than 0.1, it is considered the swing phase; when it is less than or equal to 0.1, it is considered the stance phase. 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			

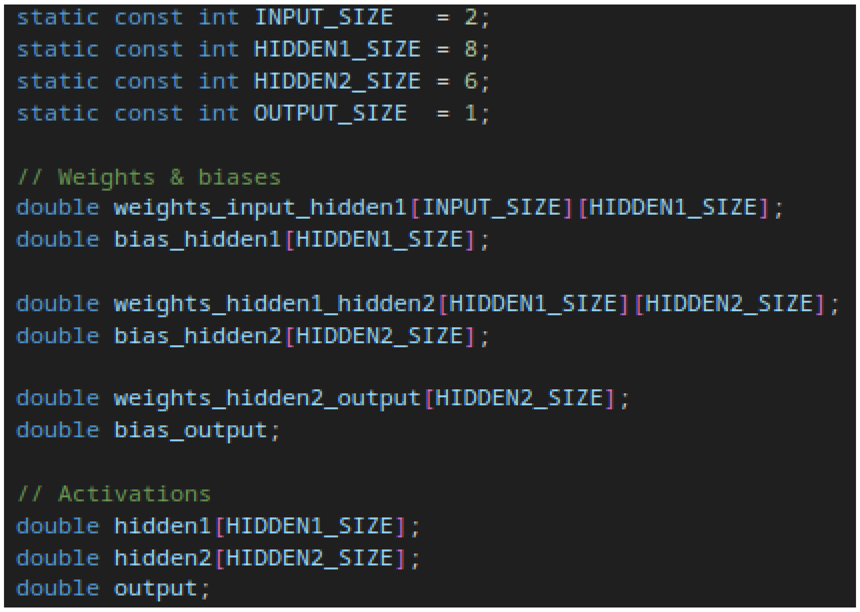
The input comes from a .txt file containing two features, X1 and X2, which represent signal parameters. Labels do not need to be provided, as the program automatically assigns them using:



```cpp
bool loadFromFile(const std::string& path) {
  std::ifstream file(path);
  if (!file.is_open()) {
    std::cerr << "Failed to open file: " << path << std::endl;
    return false;
  }

  std::string line;
  while (std::getline(file, line)) {
    std::istringstream iss(line);
    double             x1, x2;
    if (iss >> x1 >> x2) {
      int label = (x1 < 0.1 && x2 < 0.1) ? 0 : 1;  // ✅ Auto-label
      data.push_back({x1, x2, label});
    }
  }

  file.close();
  return true;
}
```

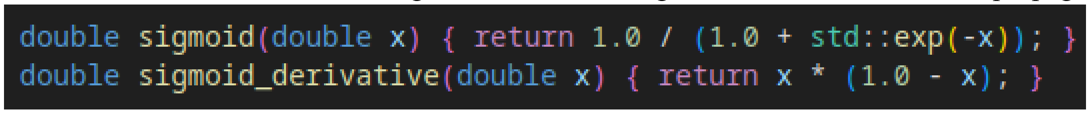
this algorithm is looking if any data > 0.1 then it will labeled 0 or stance

The model architecture uses a Multi-Layer Perceptron (MLP) with the following structure: 2 input neurons, 2 hidden layers (8 neurons in the first hidden layer and 6 in the second), and 1 output neuron with a value between 0 and 1. The use of two hidden layers allows the model to balance learning capacity and computational efficiency, making it suitable for small to medium-sized datasets



```cpp
static const int INPUT_SIZE   = 2;
static const int HIDDEN1_SIZE = 8;
static const int HIDDEN2_SIZE = 6;
static const int OUTPUT_SIZE  = 1;

// Weights & biases
double weights_input_hidden1[INPUT_SIZE][HIDDEN1_SIZE];
double bias_hidden1[HIDDEN1_SIZE];

double weights_hidden1_hidden2[HIDDEN1_SIZE][HIDDEN2_SIZE];
double bias_hidden2[HIDDEN2_SIZE];

double weights_hidden2_output[HIDDEN2_SIZE];
double bias_output;

// Activations
double hidden1[HIDDEN1_SIZE];
double hidden2[HIDDEN2_SIZE];
double output;
```

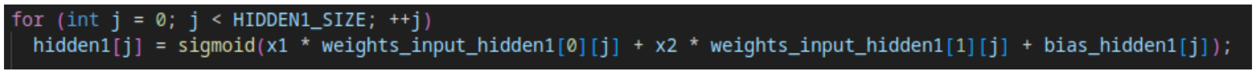
Variabel that using int it will be constant because it uses on total node, total loop and etc. when the doubled variable is for accuration so not missing single number that make it different.
The activation function used here is the sigmoid function, along with its derivative for backpropagation.



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

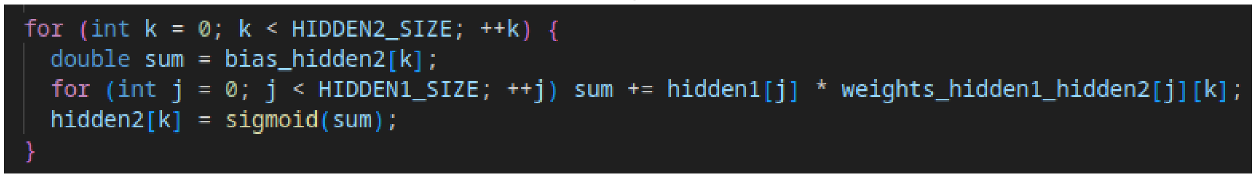
After that, the feedforward process begins, with the first hidden layer using the following formula:

$$HiddenLayer1_j = \sigma\left(\sum_{i=1}^{2} x_i \, . \, w_{ij}^{(1)} + b_j^{(1)}\right)$$



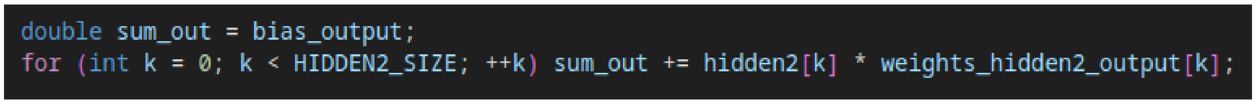
```
for (int j = 0; j < HIDDEN1_SIZE; ++j)
  hidden1[j] = sigmoid(x1 * weights_input_hidden1[0][j] + x2 * weights_input_hidden1[1][j] + bias_hidden1[j]);
```

$$HiddenLayer2_j = \sigma\left(\sum_{j=1}^{8} h_j^{(1)} . \, w_{jk}^{(2)} + b_k^{(2)}\right)$$



```
for (int k = 0; k < HIDDEN2_SIZE; ++k) {
  double sum = bias_hidden2[k];
  for (int j = 0; j < HIDDEN1_SIZE; ++j) sum += hidden1[j] * weights_hidden1_hidden2[j][k];
  hidden2[k] = sigmoid(sum);
}
```

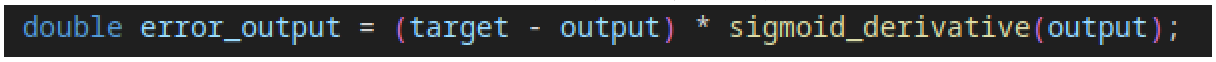
$$Output = \sigma\left(\sum_{k=1}^{6} h_k^{(2)} . \, w_k^{(3)} + b^{(3)}\right)$$



```
double sum_out = bias_output;
for (int k = 0; k < HIDDEN2_SIZE; ++k) sum_out += hidden2[k] * weights_hidden2_output[k];
```

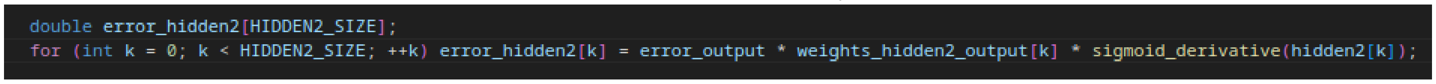
Once the feedforward process is complete, the output delta is used to update the bias — a process known as backpropagation — using the following formula and implementation:

$$\delta output = (y - \hat{y}) \, . \, \sigma^{-1}(\hat{y})$$



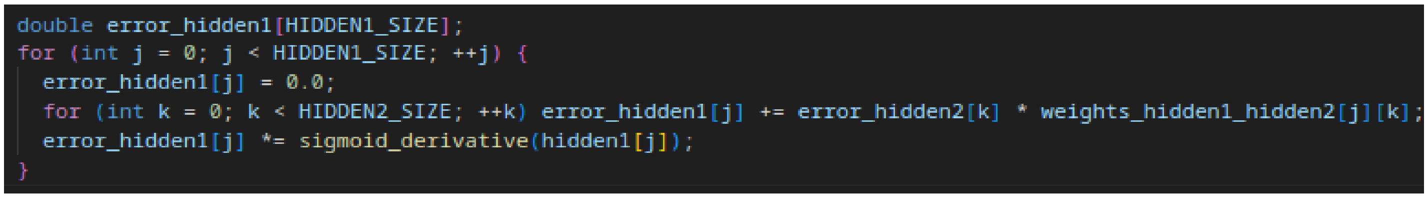
```
double error_output = (target - output) * sigmoid_derivative(output);
```

$$\delta HiddenLayerError2 = \delta output \, . \, w_{hiddeen\ layer2} \, . \, \sigma^{-1}(HiddenLayer2)$$



```
double error_hidden2[HIDDEN2_SIZE];
for (int k = 0; k < HIDDEN2_SIZE; ++k) error_hidden2[k] = error_output * weights_hidden2_output[k] * sigmoid_derivative(hidden2[k]);
```

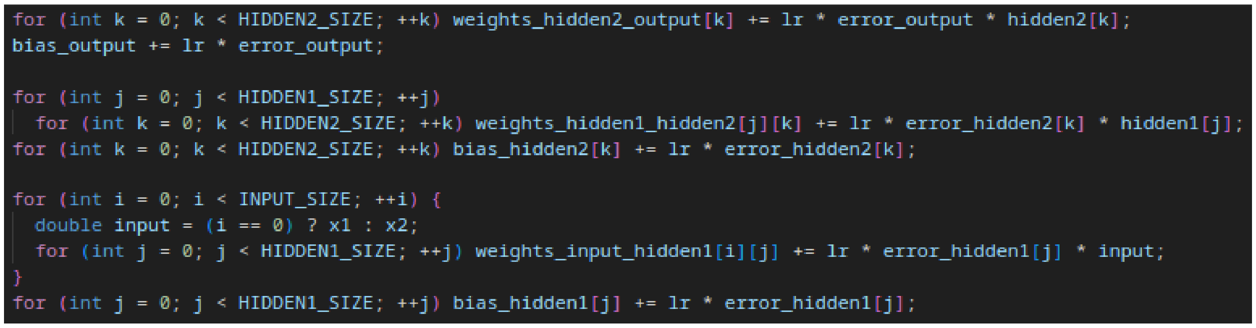
$$\delta HiddenLayerError1_j = \left(\sum_k \delta HiddenLayerError2_k \, . \, w_{jk}^{(2)}\right) . \, \sigma^{-1}(HiddenLayer1)$$



```
double error_hidden1[HIDDEN1_SIZE];
for (int j = 0; j < HIDDEN1_SIZE; ++j) {
  error_hidden1[j] = 0.0;
  for (int k = 0; k < HIDDEN2_SIZE; ++k) error_hidden1[j] += error_hidden2[k] * weights_hidden1_hidden2[j][k];
  error_hidden1[j] *= sigmoid_derivative(hidden1[j]);
}
```

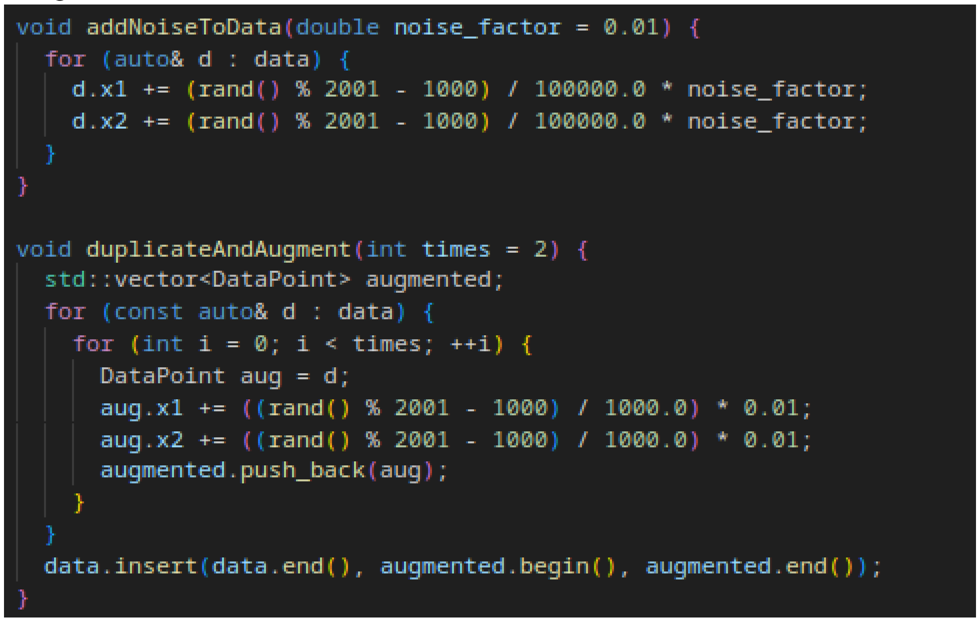
This allows the weights to be updated in order to reach the desired output value

$$Weight \leftarrow w + \eta \, . \, \delta \, . \, input$$



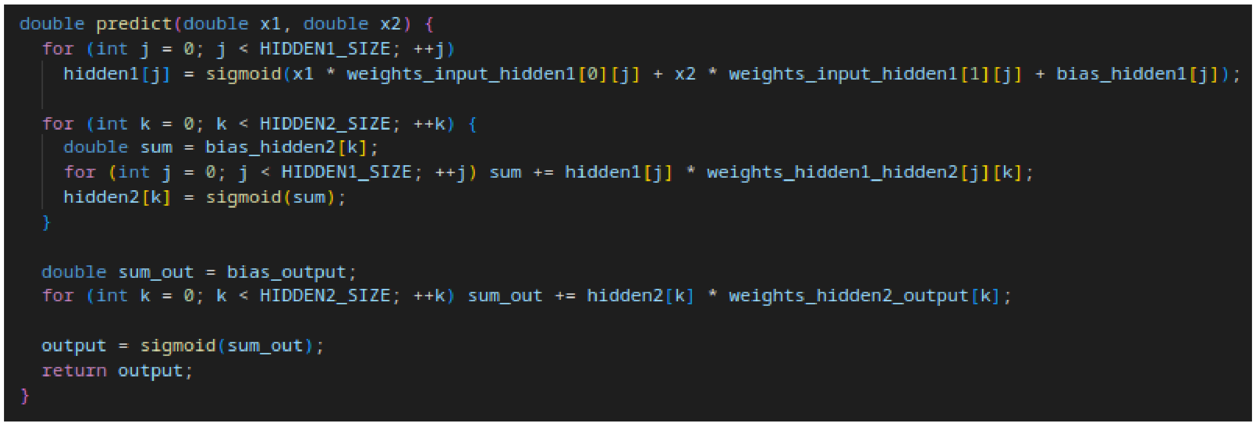
```cpp
for (int k = 0; k < HIDDEN2_SIZE; ++k) weights_hidden2_output[k] += lr * error_output * hidden2[k];
bias_output += lr * error_output;

for (int j = 0; j < HIDDEN1_SIZE; ++j)
  for (int k = 0; k < HIDDEN2_SIZE; ++k) weights_hidden1_hidden2[j][k] += lr * error_hidden2[k] * hidden1[j];
for (int k = 0; k < HIDDEN2_SIZE; ++k) bias_hidden2[k] += lr * error_hidden2[k];

for (int i = 0; i < INPUT_SIZE; ++i) {
  double input = (i == 0) ? x1 : x2;
  for (int j = 0; j < HIDDEN1_SIZE; ++j) weights_input_hidden1[i][j] += lr * error_hidden1[j] * input;
}
for (int j = 0; j < HIDDEN1_SIZE; ++j) bias_hidden1[j] += lr * error_hidden1[j];
```

With the prediction result, and by adding data augmentation, the dataset becomes more varied. This is achieved through functions such as addNoiseToData and duplicateAndAugment, which are applied before and after training

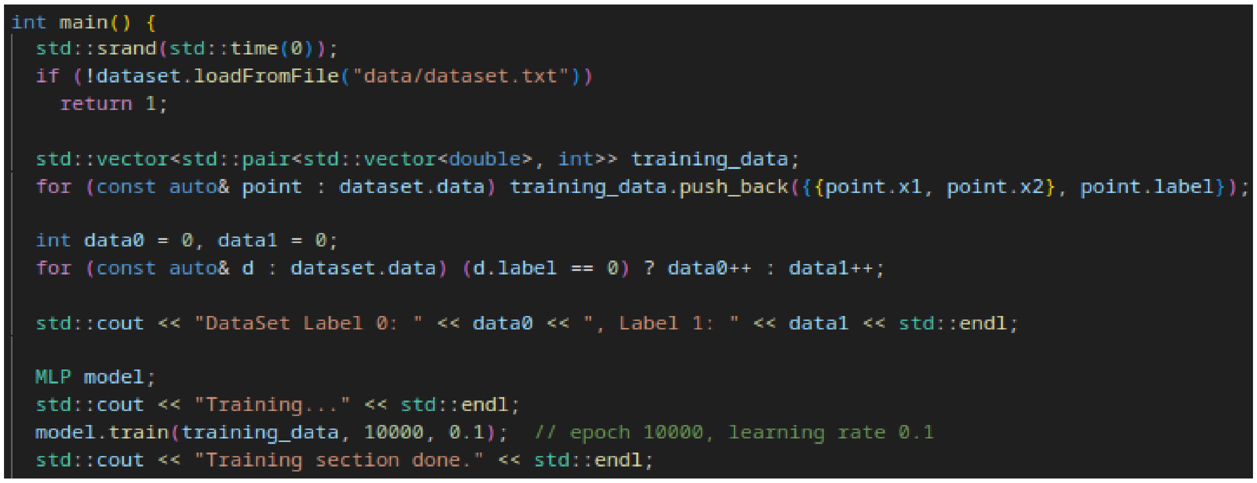


```cpp
void addNoiseToData(double noise_factor = 0.01) {
  for (auto& d : data) {
    d.x1 += (rand() % 2001 - 1000) / 100000.0 * noise_factor;
    d.x2 += (rand() % 2001 - 1000) / 100000.0 * noise_factor;
  }
}

void duplicateAndAugment(int times = 2) {
  std::vector<DataPoint> augmented;
  for (const auto& d : data) {
    for (int i = 0; i < times; ++i) {
      DataPoint aug = d;
      aug.x1 += ((rand() % 2001 - 1000) / 1000.0) * 0.01;
      aug.x2 += ((rand() % 2001 - 1000) / 1000.0) * 0.01;
      augmented.push_back(aug);
    }
  }
  data.insert(data.end(), augmented.begin(), augmented.end());
}
```

This section is the implementation of the feedforward algorithm for a neural network model (MLP) that takes two inputs and produces one output based on weights and biases that have been initialized and learned during training.

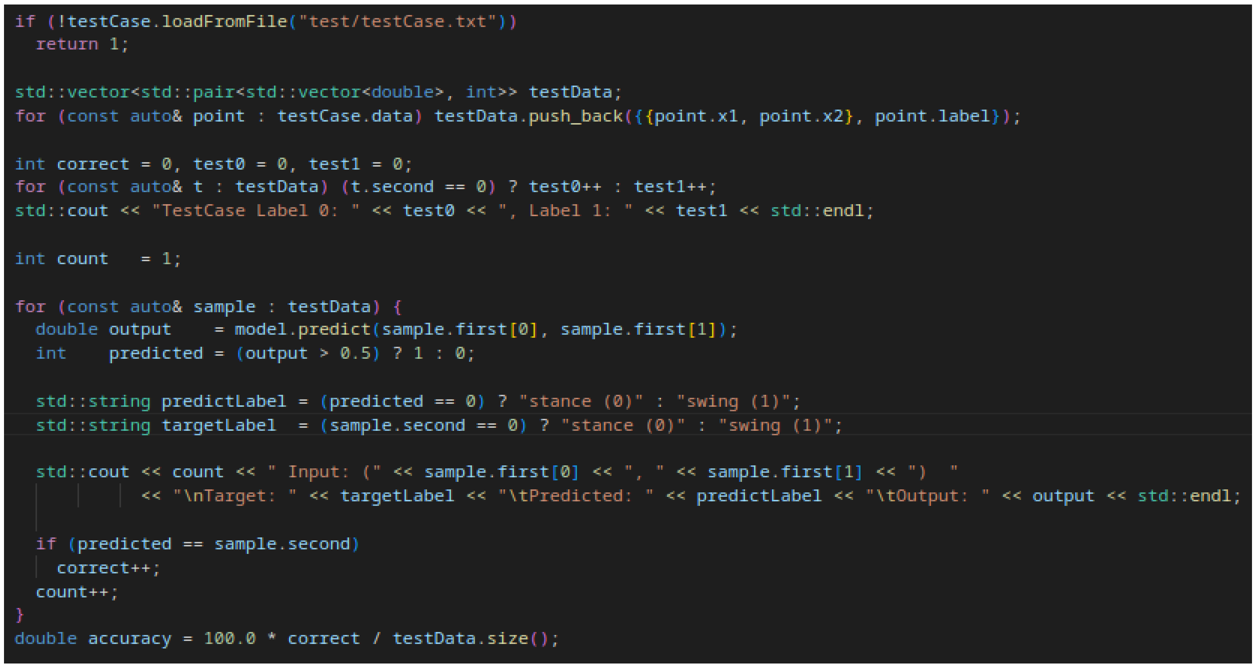


```cpp
double predict(double x1, double x2) {
  for (int j = 0; j < HIDDEN1_SIZE; ++j)
    hidden1[j] = sigmoid(x1 * weights_input_hidden1[0][j] + x2 * weights_input_hidden1[1][j] + bias_hidden1[j]);

  for (int k = 0; k < HIDDEN2_SIZE; ++k) {
    double sum = bias_hidden2[k];
    for (int j = 0; j < HIDDEN1_SIZE; ++j) sum += hidden1[j] * weights_hidden1_hidden2[j][k];
    hidden2[k] = sigmoid(sum);
  }

  double sum_out = bias_output;
  for (int k = 0; k < HIDDEN2_SIZE; ++k) sum_out += hidden2[k] * weights_hidden2_output[k];

  output = sigmoid(sum_out);
  return output;
}
```



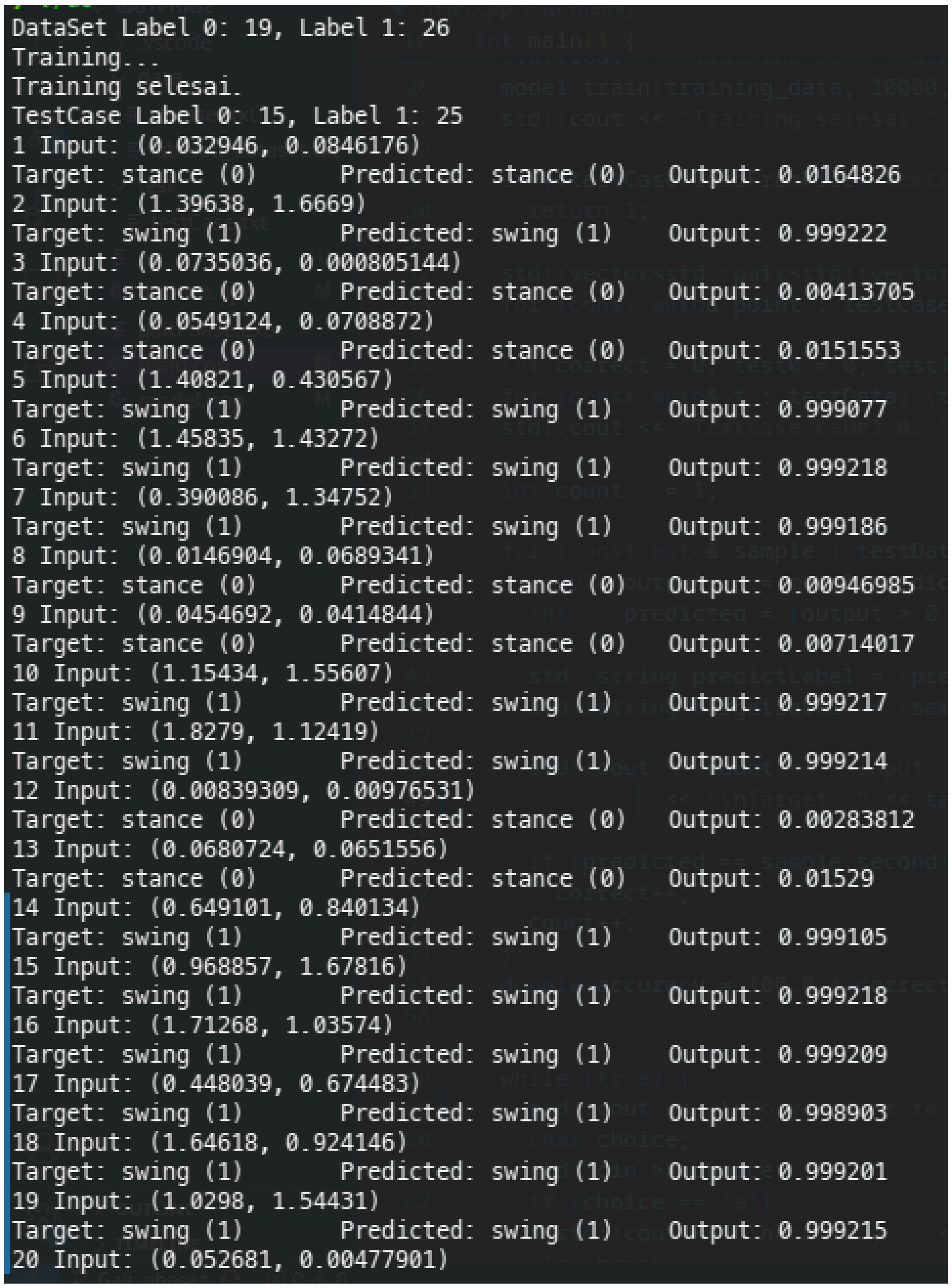
```cpp
int main() {
  std::srand(std::time(0));
  if (!dataset.loadFromFile("data/dataset.txt"))
    return 1;

  std::vector<std::pair<std::vector<double>, int>> training_data;
  for (const auto& point : dataset.data) training_data.push_back({{point.x1, point.x2}, point.label});

  int data0 = 0, data1 = 0;
  for (const auto& d : dataset.data) (d.label == 0) ? data0++ : data1++;

  std::cout << "DataSet Label 0: " << data0 << ", Label 1: " << data1 << std::endl;

  MLP model;
  std::cout << "Training..." << std::endl;
  model.train(training_data, 10000, 0.1);  // epoch 10000, learning rate 0.1
  std::cout << "Training section done." << std::endl;
```

this is the main program for training the model

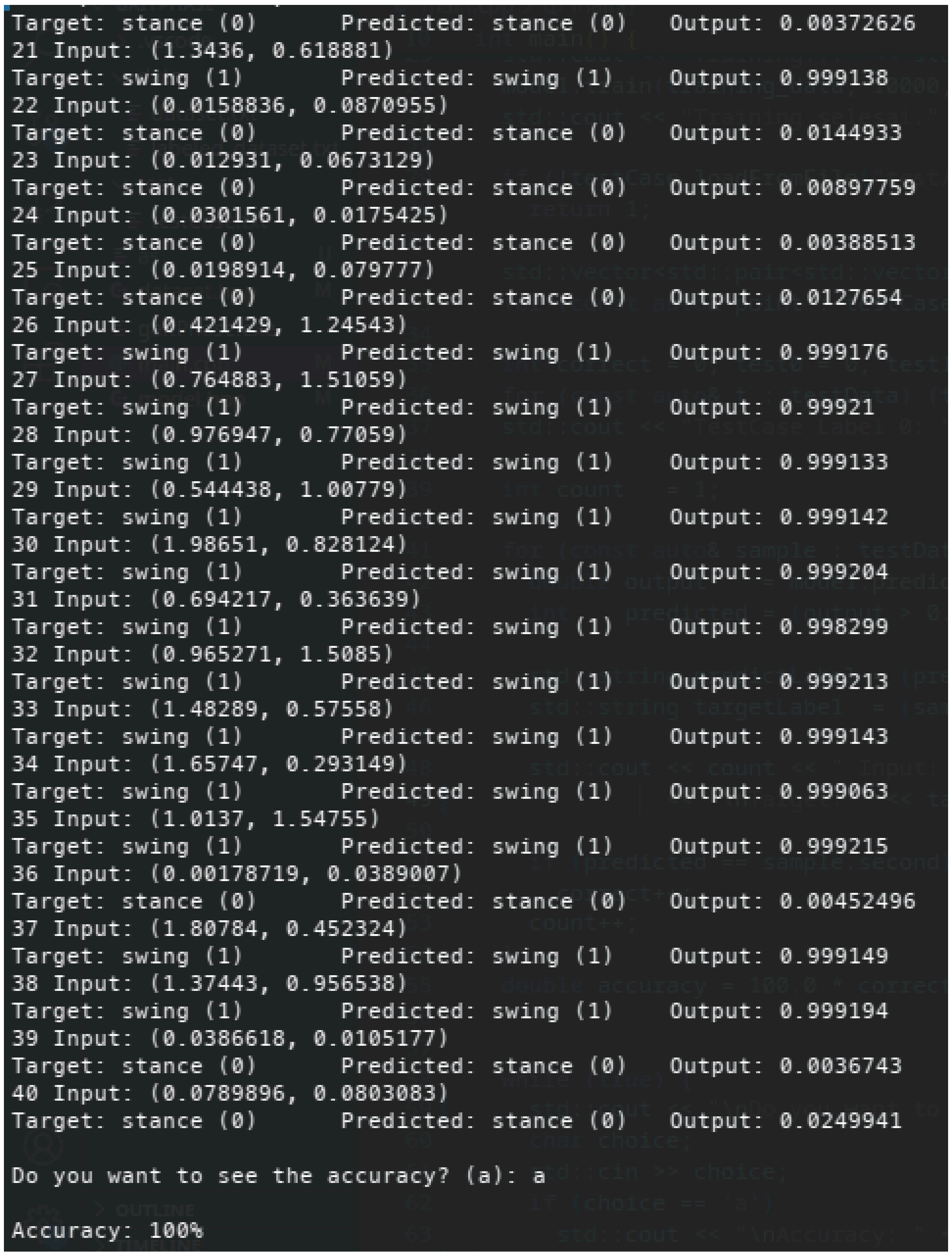


```cpp
if (!testCase.loadFromFile("test/testCase.txt"))
  return 1;

std::vector<std::pair<std::vector<double>, int>> testData;
for (const auto& point : testCase.data) testData.push_back({{point.x1, point.x2}, point.label});

int correct = 0, test0 = 0, test1 = 0;
for (const auto& t : testData) (t.second == 0) ? test0++ : test1++;
std::cout << "TestCase Label 0: " << test0 << ", Label 1: " << test1 << std::endl;

int count   = 1;

for (const auto& sample : testData) {
  double output    = model.predict(sample.first[0], sample.first[1]);
  int    predicted = (output > 0.5) ? 1 : 0;

  std::string predictLabel = (predicted == 0) ? "stance (0)" : "swing (1)";
  std::string targetLabel  = (sample.second == 0) ? "stance (0)" : "swing (1)";

  std::cout << count << " Input: (" << sample.first[0] << ", " << sample.first[1] << ")  "
            << "\nTarget: " << targetLabel << "\tPredicted: " << predictLabel << "\tOutput: " << output << std::endl;

  if (predicted == sample.second)
    correct++;
  count++;
}
double accuracy = 100.0 * correct / testData.size();
```

This is a program to view the results of its classification label prediction.



```
DataSet Label 0: 19, Label 1: 26
Training...
Training selesai.
TestCase Label 0: 15, Label 1: 25
1 Input: (0.032946, 0.0846176)
Target: stance (0)      Predicted: stance (0)   Output: 0.0164826
2 Input: (1.39638, 1.6669)
Target: swing (1)       Predicted: swing (1)    Output: 0.999222
3 Input: (0.0735036, 0.000805144)
Target: stance (0)      Predicted: stance (0)   Output: 0.00413705
4 Input: (0.0549124, 0.0708872)
Target: stance (0)      Predicted: stance (0)   Output: 0.0151553
5 Input: (1.40821, 0.430567)
Target: swing (1)       Predicted: swing (1)    Output: 0.999077
6 Input: (1.45835, 1.43272)
Target: swing (1)       Predicted: swing (1)    Output: 0.999218
7 Input: (0.390086, 1.34752)
Target: swing (1)       Predicted: swing (1)    Output: 0.999186
8 Input: (0.0146904, 0.0689341)
Target: stance (0)      Predicted: stance (0)   Output: 0.00946985
9 Input: (0.0454692, 0.0414844)
Target: stance (0)      Predicted: stance (0)   Output: 0.00714017
10 Input: (1.15434, 1.55607)
Target: swing (1)       Predicted: swing (1)    Output: 0.999217
11 Input: (1.8279, 1.12419)
Target: swing (1)       Predicted: swing (1)    Output: 0.999214
12 Input: (0.00839309, 0.00976531)
Target: stance (0)      Predicted: stance (0)   Output: 0.00283812
13 Input: (0.0680724, 0.0651556)
Target: stance (0)      Predicted: stance (0)   Output: 0.01529
14 Input: (0.649101, 0.840134)
Target: swing (1)       Predicted: swing (1)    Output: 0.999105
15 Input: (0.968857, 1.67816)
Target: swing (1)       Predicted: swing (1)    Output: 0.999218
16 Input: (1.71268, 1.03574)
Target: swing (1)       Predicted: swing (1)    Output: 0.999209
17 Input: (0.448039, 0.674483)
Target: swing (1)       Predicted: swing (1)    Output: 0.998903
18 Input: (1.64618, 0.924146)
Target: swing (1)       Predicted: swing (1)    Output: 0.999201
19 Input: (1.0298, 1.54431)
Target: swing (1)       Predicted: swing (1)    Output: 0.999215
20 Input: (0.052681, 0.00477901)
```



```
Target: stance (0)      Predicted: stance (0)   Output: 0.00372626
21 Input: (1.3436, 0.618881)
Target: swing (1)       Predicted: swing (1)    Output: 0.999138
22 Input: (0.0158836, 0.0870955)
Target: stance (0)      Predicted: stance (0)   Output: 0.0144933
23 Input: (0.012931, 0.0673129)
Target: stance (0)      Predicted: stance (0)   Output: 0.00897759
24 Input: (0.0301561, 0.0175425)
Target: stance (0)      Predicted: stance (0)   Output: 0.00388513
25 Input: (0.0198914, 0.079777)
Target: stance (0)      Predicted: stance (0)   Output: 0.0127654
26 Input: (0.421429, 1.24543)
Target: swing (1)       Predicted: swing (1)    Output: 0.999176
27 Input: (0.764883, 1.51059)
Target: swing (1)       Predicted: swing (1)    Output: 0.99921
28 Input: (0.976947, 0.77059)
Target: swing (1)       Predicted: swing (1)    Output: 0.999133
29 Input: (0.544438, 1.00779)
Target: swing (1)       Predicted: swing (1)    Output: 0.999142
30 Input: (1.98651, 0.828124)
Target: swing (1)       Predicted: swing (1)    Output: 0.999204
31 Input: (0.694217, 0.363639)
Target: swing (1)       Predicted: swing (1)    Output: 0.998299
32 Input: (0.965271, 1.5085)
Target: swing (1)       Predicted: swing (1)    Output: 0.999213
33 Input: (1.48289, 0.57558)
Target: swing (1)       Predicted: swing (1)    Output: 0.999143
34 Input: (1.65747, 0.293149)
Target: swing (1)       Predicted: swing (1)    Output: 0.999063
35 Input: (1.0137, 1.54755)
Target: swing (1)       Predicted: swing (1)    Output: 0.999215
36 Input: (0.00178719, 0.0389007)
Target: stance (0)      Predicted: stance (0)   Output: 0.00452496
37 Input: (1.80784, 0.452324)
Target: swing (1)       Predicted: swing (1)    Output: 0.999149
38 Input: (1.37443, 0.956538)
Target: swing (1)       Predicted: swing (1)    Output: 0.999194
39 Input: (0.0386618, 0.0105177)
Target: stance (0)      Predicted: stance (0)   Output: 0.0036743
40 Input: (0.0789896, 0.0803083)
Target: stance (0)      Predicted: stance (0)   Output: 0.0249941

Do you want to see the accuracy? (a): a

Accuracy: 100%
```

Output from program using test case that adding noise 40% from train dataset and 60% random number

3. Conclusion

From this project, it can be concluded that a Multi-Layer Perceptron (MLP) neural network is an effective and efficient method for classifying gait phases, specifically identifying the stance and swing phases of a gait cycle. The use of only two input features representing foot motion or position, combined with an automatic labeling rule based on a simple threshold, allows for consistent and annotation-free data processing.

The chosen MLP architecture—consisting of two input neurons, two hidden layers (with 8 and 6 neurons), and one output neuron—achieves a balance between performance and computational efficiency, making it suitable for smaller datasets. The model was trained using the Error Backpropagation Algorithm (EBPA), with sigmoid activation functions enabling it to learn complex, non-linear patterns. Despite common challenges like vanishing gradients, these were mitigated through normalization and careful weight initialization.

Furthermore, data augmentation techniques such as noise addition and data duplication significantly improved the model's robustness and generalization ability. Testing with 40% noise augmentation demonstrated that the model could maintain its predictive accuracy, highlighting its reliability even under less-than-ideal data conditions.