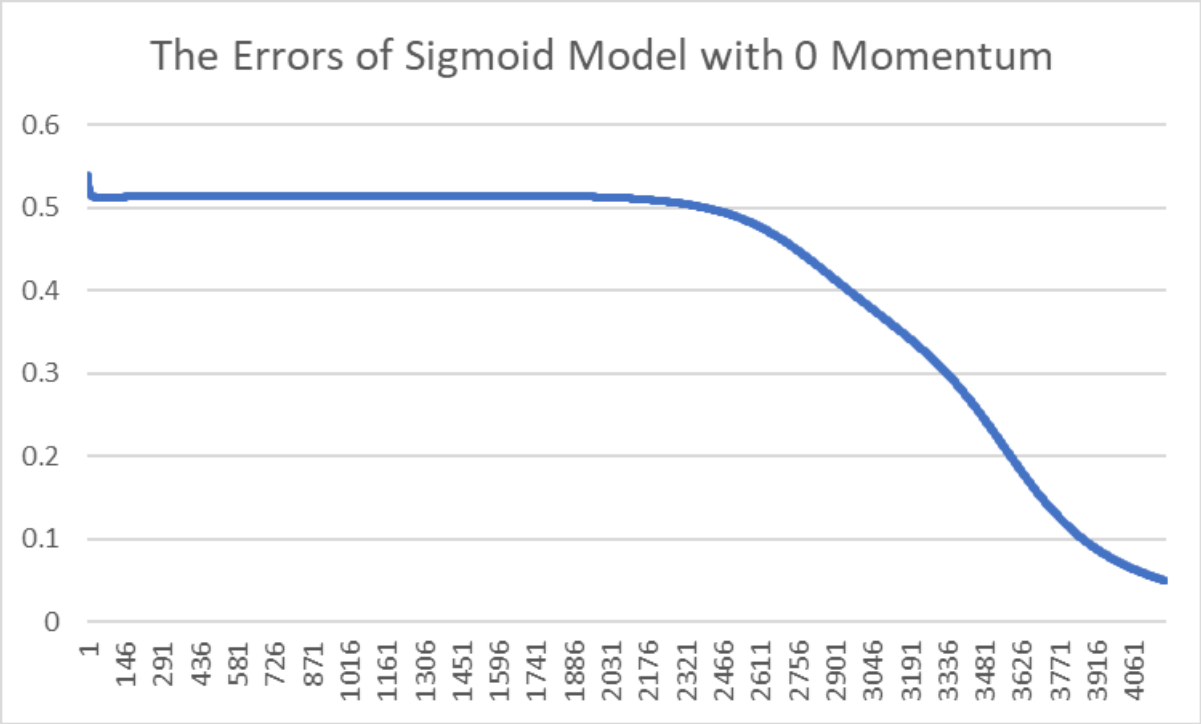
Global setting

The input dimension is configured as 2, with 4 hidden neurons and an output dimension of 1. The input dataset used for training corresponds to the XOR problem, with values {0, 1} when using the sigmoid activation function and {-1, 1} when using the bipolar activation function. The initial weights are randomly assigned within the range of -0.5 to 0.5. Throughout the training process, a learning rate of 0.2 is applied.

Part a

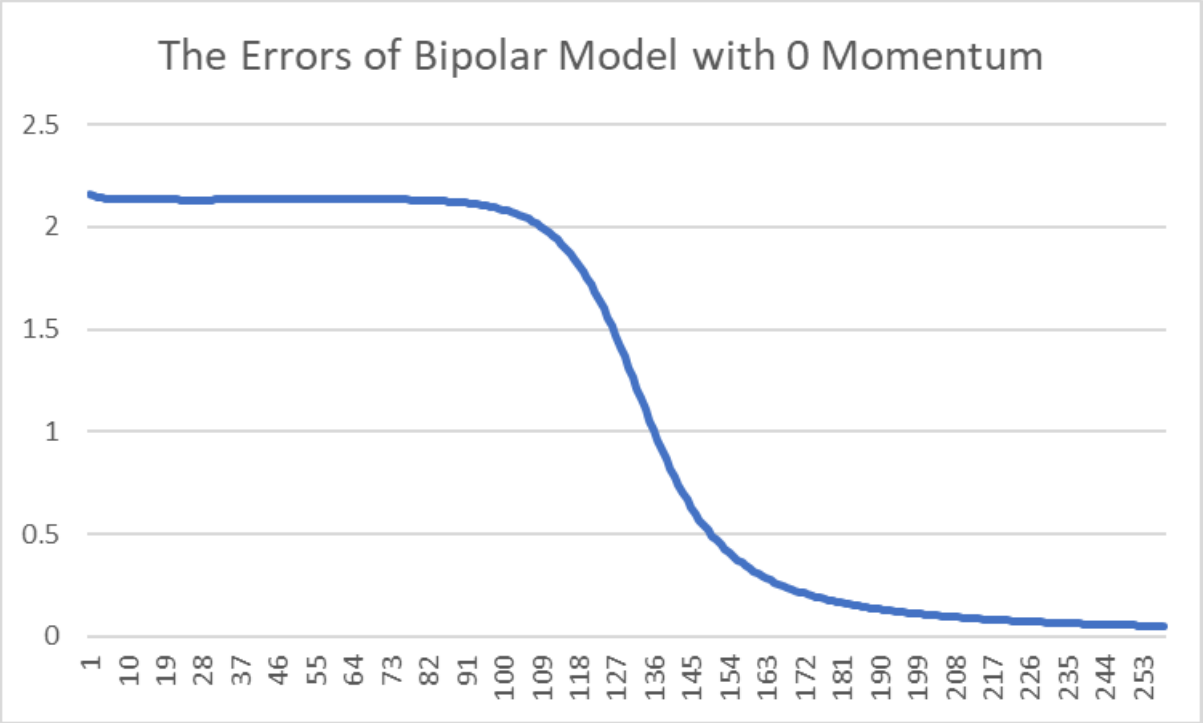
In this section, the model referred to as **"sigmoid_noM"** utilizes the sigmoid function without momentum. On average, across 1000 trials, this model took approximately **4065** epochs to achieve an error of 0.05. Plot 1 illustrates the training progress for one of these sample trials of the "sigmoid_noM" model.



Plot 1: the error plot of sigmoid_noM

Part b

Now, the activation function has been switched to Bipolar, and the training dataset has also been updated to represent false and true as {-1, 1}. The current model is named "**bipolar_noM**," indicating the use of the Bipolar activation function without momentum. On average, it takes approximately **253** epochs for this model to reach an error of 0.05, which is only 1/16th of the time it took when using the sigmoid function. Below is Plot 2, which displays the training progress for one sample trial of the "bipolar_noM" model:

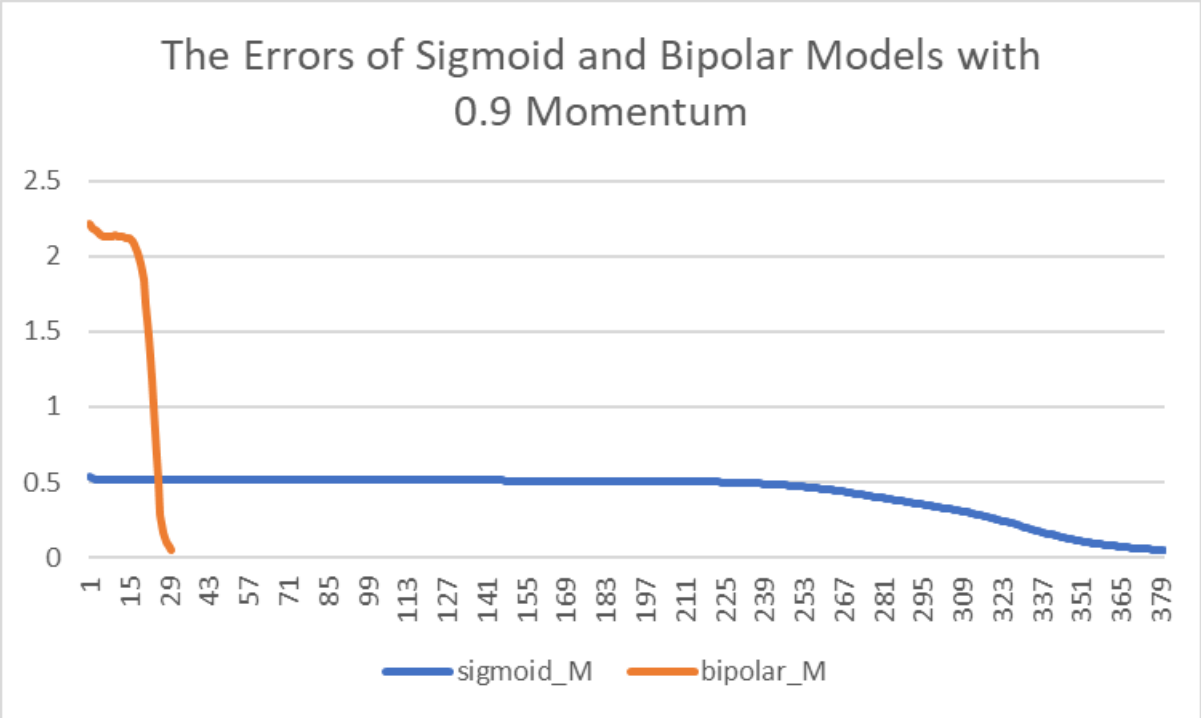


Plot 2: the error plot of bipolar_noM

Part c

Then, I adjusted the momentum to 0.9 for both types of activation functions to ensure a fair comparison. Two models were created: "**sigmoid_M**," using the sigmoid activation function with a momentum of 0.9, and "**bipolar_M**," employing the bipolar activation function with the same 0.9 momentum setting.

On average, it took **438** epochs to train "sigmoid_M" to achieve a 0.05 error, while "bipolar_M" reached the same error level in just **34** epochs. This represents nearly a tenfold reduction in training time compared to using the sigmoid function. The introduction of 0.9 momentum resulted in both models requiring approximately 1/10th of the epochs needed when compared to no momentum. Notably, the "bipolar_M" model with 0.9 momentum significantly outperformed the "bipolar_noM" model without momentum in part b. Consequently, choosing the Bipolar activation function with 0.9 momentum proved to be considerably faster and more efficient compared to using the sigmoid activation function and bipolar activation without momentum. Plot 3 illustrates the errors for "sigmoid_M" (depicted in blue) and "bipolar_M" (depicted in orange).



Plot 3: the error plots of sigmoid_M (blue) and bipolar_M (orange).

The concluded table of average epochs is shown below:

Section	Model name	Activation function	Momentum	Average epochs
part a	sigmoid_noM	Sigmoid	0.0	4065
part b	bipolar_noM	Bipolar	0.0	253
part c	sigmoid_M	Sigmoid	0.9	438
	bipolar_M	Bipolar	0.9	**34**

Code introduction

I have implemented several classes in my code, including Neuron, Layer, NeuralNet, writeCSV, and Main. Here's a brief overview of how each class functions:

1. **Neuron Class:**
   a) The Neuron class represents an individual neuron in the neural network.
   b) It saves the calculation and updating of error signals for the neuron.
2. **Layer Class:**
   a) The Layer class contains multiple neurons, instances of the Neuron class.
   b) The number of neurons in this layer is defined in the Main class using the variable num_hiddens.

c) The Layer class also stores the weights connecting each neuron within this layer.

3. **NeuralNet Class:**
   a) The NeuralNet class manages the neural network as a whole.
   b) It consists of multiple layers, and the number of layers is defined in the Main class using the variable num_layers.
   c) When the train method is called in this class, the network conducts both forward and backward propagations. It achieves this by invoking the corresponding forward and backward functions of each layer object within the network.
4. **writeCSV Class:**
   This class is responsible for handling the writing of data, results, or logs to CSV files.
5. **Main Class:**
   a) The Main class serves as the central control hub for the neural network.
   b) It is responsible for:
      i. Creating instances of NeuralNet objects for comparison purposes.
      ii. Orchestrating the training process, specifying the number of training epochs and trials (e.g., 10,000 epochs and 1,000 trials).

Appendix

Neuron class:

```
public class Neuron{
   private double error_signal = 0.0;
   private double output = 0.0;
   public Neuron(){
      error_signal = 0.0;
   }

   public double getError_signal(){
      return error_signal;
   }
   public void setError_signal(double newError){
      error_signal = newError;
   }
   public double getOutput(){
      return output;
   }
   public void setOutput(double new_out){
      output = new_out;
   }
}
```

Layer class:

```
import java.util.ArrayList;

public class Layer{
   // The number of hidden units in this layer. The dimension of output vector is equal to num_Neuron
   private int num_Neuron = 4;
   // The dimension of input to this layer.
   private int input_size = 2;
   private double momentum = 0.9;
   private double learning_rate = 0.1;
   //If this is the last layer
   private boolean last_layer = false;
   // Activation funciton name
   private String activation_func = "sigmoid";
   private int a = -1;
   private int b = 1;
   // A layer contains num_Neurons neurons
   public Neuron[] neurons;
   // The dimension of weights is set to num_Neurons x input_size. (H x D)
   public double[][] weights;
   public double[][] prev_weightchanges;
   // store the inputs to this layer in forward
   public double[] inputs;
   // store the outputs from this layer in forward
   public double[] outputs;


   public Layer(int num_Neuron, int input_size, boolean last_layer, double momentum, double learning_rate,
String activation_func,  int a, int b){
      this.num_Neuron = num_Neuron;
```

```
    this.input_size = input_size;
    this.last_layer = last_layer;
    this.momentum = momentum;
    this.learning_rate = learning_rate;
    this.activation_func = activation_func;
    neurons = new Neuron[num_Neuron];
    //initialize neurons
    for (int j = 0; j < num_Neuron; j++){
        neurons[j] = new Neuron();
    }
    // input_size + 1 for bias
    weights = new double[num_Neuron][input_size+1];
    prev_weightchanges = new double[num_Neuron][input_size+1];
    inputs = new double[input_size+1];
    outputs = new double[num_Neuron];
    this.a = a;
    this.b = b;
    // Initialize the values
    zero_outputs();
    zero_prevs();
    zero_inputs();
    initialize_weights();
}
public void forward(double[] new_inputs){
    update_inputs(new_inputs);
    zero_outputs();
    for(int j=0; j<num_Neuron; j++){
        for(int i=0; i<input_size+1; i++){
            outputs[j] += weights[j][i] * inputs[i];
        }
        if (activation_func.equals("sigmoid")) {
            outputs[j] = sigmoid(outputs[j]);
        }
        else {
            outputs[j] = customSigmoid(outputs[j]);
        }
        neurons[j].setOutput(outputs[j]);
    }
}
// error_signal = output[j] * (1- output[j]) * sum(next_layer.neurons[h].error_signal * next_layer.weights[h][j])
public void backward(Layer next_layer){
    for (int j=0; j<num_Neuron; j++){
        double df = 0;
        if (activation_func.equals("sigmoid")){
            df = outputs[j] * (1 - outputs[j]);
        }
        else {
            df = 0.5 * (1 - Math.pow(outputs[j], 2));
        }
        double sum_aboves = 0;
        for (int h=0; h<next_layer.getNum_Neuron(); h++){
            sum_aboves += next_layer.weights[h][j] * next_layer.neurons[h].getError_signal();
        }
        double new_err = df*sum_aboves;
        neurons[j].setError_signal(new_err);
    }
}
// error_signal = output[j] * (1- output[j]) * error[j], error[j] = label[j] - output[j]
```

```
public void backward_lastlayer(double[] errors){
    for (int j=0; j<num_Neuron; j++){
        double df;
        if (activation_func.equals("sigmoid")){
            df = outputs[j] * (1 - outputs[j]);
        }
        else {
            df = 0.5 * (1 - Math.pow(outputs[j], 2));
        }
        double r = errors[j];
        double new_err = df*r;
        neurons[j].setError_signal(new_err);
    }
}
// update weights: w[j][i] = w[j][i] + a*prev_dw[j][i] + lr*neurons[j].error_signal*inputs[i]
public void update_weights(){
    for (int j=0; j<num_Neuron; j++){
        for (int i=0; i<input_size+1; i++){
            double gradient = neurons[j].getError_signal()*inputs[i];
            double dw = momentum*prev_weightchanges[j][i] + learning_rate*gradient;
            weights[j][i] += dw;
            prev_weightchanges[j][i] = dw;
        }
    }
}
// get the new inputs from previous layer
public void update_inputs(double[] new_inputs){
    for(int i=0; i<input_size+1; i++){
        if(i == input_size){
            inputs[i] = NeuralNetInterface.bias;
        }
        else inputs[i] = new_inputs[i];
    }
}
public void zero_outputs(){
    for(int j=0; j<num_Neuron; j++){
        outputs[j] = 0;
    }
}
public void zero_inputs(){
    for(int i=0; i<input_size-1; i++){
        inputs[i] = 0;
    }
    inputs[input_size-1] = NeuralNetInterface.bias;
}
public void zero_prevs(){
    for (int j=0; j<num_Neuron; j++){
        for (int i=0; i<input_size+1; i++){
            prev_weightchanges[j][i] = 0;
        }
    }
}
// initialize random weights
public void initialize_weights(){
    for(int j = 0; j < num_Neuron; j++) {
        for(int i = 0; i < input_size+1; i++) {
            //Math.random() return 0 to 1
            weights[j][i] = Math.random() - 0.5;
```

```
        }
      }
    }
    /**
     * Return a binary sigmoid of the input X(The activation function)
     * @param x The input
     * @return f(x) = 1 / (1+e(-x))
     */

    public double sigmoid(double x) {
      return (double)1 / (1 + Math.exp(-x));
    }
    /**
     * Return a bipolar sigmoid of the input X(The activation function)
     * @param x The input
     * @return f(x) = 2 / (1+e(-x)) - 1
     */

    public double customSigmoid(double x) {
      return (double)(b - a) / (1 + Math.exp(-x)) + a;
    }
    public int getNum_Neuron(){
      return num_Neuron;
    }
    public boolean getLast_layer(){
      return last_layer;
    }
}
```

NeuralNet class:

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class NeuralNet implements NeuralNetInterface{
  private int input_size = 2;
  private int num_hiddens = 4;
  private int output_size = 1;
  private int num_layers = 2;
  private Layer[] layers;
  private double learning_rate = 0.2;
  private double momentum = 0;
  private int a = -1;
  private int b = 1;
  private String activation_func = "sigmoid";
  private double error_threshold = 0.05;
  private int epochs = 0;
  private List<Double> error_logs;
  private String file_name;

  public NeuralNet(int input_size, int num_hiddens, int output_size, int num_layers, double learning_rate,
double momentum, int a, int b, String activation_func, int epochs, String file_name) {
    this.input_size = input_size;
    this.num_hiddens = num_hiddens;
    this.output_size = output_size;
    this.num_layers = num_layers;
```

```
    layers = new Layer[num_layers];
    //Create the hidden layers
    layers[0] = new Layer(num_hiddens, input_size, false, momentum, learning_rate, activation_func, a, b);
    //Create output layer
    layers[1] = new Layer(output_size, num_hiddens, true, momentum, learning_rate, activation_func, a, b);
    this.learning_rate = learning_rate;
    this.momentum = momentum;
    this.a = a;
    this.b = b;
    this.activation_func = activation_func;
    this.epochs = epochs;
    error_logs = new ArrayList<Double>();
    this.file_name = file_name;
    //epoch = 0;
}

public void forward(double[] inputs){
    double[] x = inputs;
    for (int i = 0; i < num_layers; i++){
        layers[i].forward(x);
        x = layers[i].outputs;
    }
}
//This backward func updates all weights after updating all of the error
public void backward(double[] errors){
    //for each layer call backward (backward order int i=layer_num-1, i>=0, i--)
    for (int i = num_layers-1; i>=0; i--){
        if (i == num_layers-1 && layers[i].getLast_layer()){
            layers[i].backward_lastlayer(errors);
        }
        else {
            layers[i].backward(layers[i + 1]);
        }
    }
    //for each layer call update_weights
    for (int i = num_layers-1; i>=0; i--){
        layers[i].update_weights();
    }
}

//This backward func updates weights of the layer after updating the error of the layer
public void backward2(double[] errors){
    //for each layer call backward (backward order int i=layer_num-1, i>=0, i--) and update_weights
    for (int i = num_layers-1; i>=0; i--){
        if (i == num_layers-1 && layers[i].getLast_layer()){
            layers[i].backward_lastlayer(errors);
            layers[i].update_weights();
        }
        else {
            layers[i].backward(layers[i + 1]);
            layers[i].update_weights();
        }
    }
}

// for each epoch, feed the train data to model and calculate the errors and then do the backpropagation
public int train(double[][] inputs, double[][] labels, int len_dataset){
    double total_error = 0.0;
```

```
    double[] d_error = new double[output_size];
    for (int e = 0; e < epochs; e++){
        //reset loss value
        total_error = 0.0;
        for (int i = 0; i < len_dataset; i++){
            forward(inputs[i]);
            for (int j = 0; j < output_size; j++){
                //d_error[j] = Math.sqrt(Math.pow(labels[j][0] - layers[num_layers-1].outputs[j], 2));
                d_error[j] = labels[i][j] - layers[num_layers-1].outputs[j];
                total_error += Math.pow(d_error[j], 2);
            }
            backward2(d_error);
        }
        total_error /= 2;
        //System.out.println("loss at epoch " + e + "is: " + total_error);
        error_logs.add(total_error);
        if (total_error < error_threshold){
            return e;
        }
    }
    return epochs;
}
/**
 * Return a binary sigmoid of the input X(The activation function)
 * @param x The input
 * @return f(x) = 1 / (1+e(-x))
 */
@Override
public double sigmoid(double x) {
    return (double)1 / (1 + Math.exp(-x));
}
/**
 * Return a bipolar sigmoid of the input X(The activation function)
 * @param x The input
 * @return f(x) = 2 / (1+e(-x)) - 1
 */
@Override
public double customSigmoid(double x) {
    return (double)(b - a) / (1 + Math.exp(-x)) + a;
}
/**
 * Initialization step 2 : Initialize the weights to random values.
 * For say 2 inputs, the input vector is [0] & [1]. We add [2] for the bias.
 * Like wise for hidden units. For say 2 hidden units which are stored in an array.
 * [0] & [1] are the hidden & [2] the bias.
 * We also initialise the last weight change arrays. This is to implement the alpha term.
 */
@Override
public void initializeWeights() {
    for (int l = 0; l < num_layers; l++){
        layers[l].initialize_weights();
    }
}

@Override
public void zeroWeights(){
    //
}
```

```
  @Override
  public double train(double [] X, double argValue){
     return 0.0;
  }
  @Override
  public double outputFor(double [] X){
     return 0.0;
  }
  @Override
  public void save(File argFile){

  }
  @Override
  public void load(String argFileName) throws IOException{

  }

  public void write_file(){
     String[][] content = new String[error_logs.size()][2];
     System.out.println("the written csv is " + error_logs.size() + "long.");
     for (int i = 0; i < error_logs.size(); i++){
        content[i][0] = String.valueOf(i);
        content[i][1] = String.valueOf(error_logs.get(i));
     }
     try {
        writeCSV.write(content, file_name);
     }
     catch (Exception e){
        System.out.println("Write file error");
     }
  }

}
```

main class:

```
public class main{
  public static boolean save_file = true;
  public static int trails = 1000;
  public static int epochs = 10000;
  //public static NeuralNet nn;
  public static void main(String args[]){
     int input_size = 2;
     int num_hiddens = 4;
     int output_size = 1;
     int num_layers = 2;
     double learning_rate = 0.2;
     int a = -1;
     int b = 1;

     //Dataset generating.....................
     double[][] X_sig;
     double[][] y_sig;
     double[][] X_bi;
     double[][] y_bi;
     X_sig = new double[][]{{0, 0}, {0, 1}, {1, 0}, {1, 1}};
     y_sig = new double[][]{{0}, {1}, {1}, {0}};
     X_bi = new double[][]{{-1, -1}, {-1, 1}, {1, -1}, {1, 1}};
     y_bi = new double[][]{{-1}, {1}, {1}, {-1}};
```

```
        //Dataset generating.....................

        //Create models for tests.....................
        NeuralNet nn_sigmoid_noM = new NeuralNet(input_size,num_hiddens, output_size, num_layers,
learning_rate, 0.0, a, b, "sigmoid", epochs, "sigmoid_noM");
        NeuralNet nn_bipolar_noM = new NeuralNet(input_size,num_hiddens, output_size, num_layers,
learning_rate, 0.0, a, b, "bipolar", epochs, "bipolar_noM");
        NeuralNet nn_sigmoid_M = new NeuralNet(input_size,num_hiddens, output_size, num_layers, learning_rate,
0.9, a, b, "sigmoid", epochs, "sigmoid_M");
        NeuralNet nn_bipolar_M = new NeuralNet(input_size,num_hiddens, output_size, num_layers, learning_rate,
0.9, a, b, "bipolar", epochs, "bipolar_M");
        //Create models for tests.....................

        //Training and evaluation.....................
        int total_epochs_snoM = 0;
        int total_epochs_bnoM = 0;
        int total_epochs_sM = 0;
        int total_epochs_bM = 0;
        for(int t = 0; t < trails; t++) {
            total_epochs_snoM += nn_sigmoid_noM.train(X_sig, y_sig, 4);
            total_epochs_bnoM += nn_bipolar_noM.train(X_bi,y_bi,4);
            total_epochs_sM += nn_sigmoid_M.train(X_sig,y_sig,4);
            total_epochs_bM += nn_bipolar_M.train(X_bi,y_bi,4);
            if (save_file) {
                nn_sigmoid_noM.write_file();
                nn_bipolar_noM.write_file();
                nn_sigmoid_M.write_file();
                nn_bipolar_M.write_file();
            }
            nn_sigmoid_noM.initializeWeights();
            nn_bipolar_noM.initializeWeights();
            nn_sigmoid_M.initializeWeights();
            nn_bipolar_M.initializeWeights();
        }
        System.out.println("Average epoch number using sigmoid function without momentum: " +
total_epochs_snoM/trails);
        System.out.println("Average epoch number using bipolar function without momentum: " +
total_epochs_bnoM/trails);
        System.out.println("Average epoch number using sigmoid function with momentum: " +
total_epochs_sM/trails);
        System.out.println("Average epoch number using bipolar function with momentum: " +
total_epochs_bM/trails);

        //Training and evaluation.....................
    }
}
```

writeCSV class:

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

/**
* Created by 37919 on 2023/9/30.
*/
public class writeCSV {
    public static void write(String[][] content, String out_name) throws IOException {
```

```
        File csvFile = new File(out_name += ".csv");
        FileWriter fileWriter = new FileWriter(csvFile);

        for (String[] data : content) {
            StringBuilder line = new StringBuilder();
            for (int i = 0; i < data.length; i++) {
                line.append("\"");
                line.append(data[i].replaceAll("\"","\"\""));
                line.append("\"");
                if (i != data.length - 1) {
                    line.append(',');
                }
            }
            line.append("\n");
            fileWriter.write(line.toString());
        }
        fileWriter.close();
    }
}
```