**PRACTICAL NO. 02**

**Aim :**

Placement train prediction and visual analytics for engineering college.

**Theory** **:**

This project presents a system for analyzing and predicting placement trends in engineering colleges using deep learning techniques. The system takes raw placement data, visualizes the historical trends, and enables predictive analytics to forecast future placements based on past records. The underlying model is built using a feed-forward neural network, which learns patterns from years of placement data to generate future predictions. A graphical interface makes it easy for users to interact with the system and interpret results via bar graphs.

**Working Principle** **:**

* Machine Learning Model : Multi-Layer Perceptron (MLP) Neural Network

The code implements a 3-layer feedforward neural network (also called Multi-Layer Perceptron) with the following architecture :

* Network Architecture :Input Layer (3 neurons) → Hidden Layer 1 (10 neurons) → Hidden Layer 2 (5 neurons) → Output Layer (1 neuron)
* Model Parameters : Input Features (3 parameters)
* Normalized Year → (current\_year - first\_year) / 10.0 - Represents the temporal position.
* Normalized Current Count → current\_placement\_count / max\_count - Current year's placement numbers.
* Normalized Trend → (current\_count - previous\_count) / max\_count - Rate of change from previous year.
* Network Layers :
  + Input Size → 3 neurons
  + Hidden Layer 1 → 10 neurons (with ReLU activation)
  + Hidden Layer 2 → 5 neurons (with ReLU activation)
  + Output Layer → 1 neuron (with Sigmoid activation)
* Activation Functions :

ReLU (Rectified Linear Unit): Used in hidden layers

|  |
| --- |
| relu(x) = max(0, x) |

* Sigmoid : Used in output layer to bound predictions between 0 and 1

|  |
| --- |
| sigmoid(x) = 1 / (1 + e^(-x)) |

* Training Parameters :
  + Learning Rate : 0.1
  + Epochs : 1000 iterations
  + Loss Function : Mean Squared Error (MSE)
  + Optimization : Gradient Descent with Backpropagation
* Weight Initialization :

|  |
| --- |
| scale = sqrt(2.0 / number\_of\_inputs)  weight = random\_gaussian \* scale |

**Program** **Code** **:**

import java.awt.\*;

import java.io.\*;

import java.util.\*;

import java.util.List;

import javax.swing.\*;

public class PlacementTrendAnalyzer extends JFrame {

    private JPanel mainPanel;

    private CardLayout cardLayout;

    private GraphPanel graphPanel;

    private JPanel endScreen;

    public PlacementTrendAnalyzer() {

        setTitle("Placement Bar Graph with Deep Learning");

        setSize(800, 600);

        setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

        setLocationRelativeTo(null);

        // Use CardLayout to switch between screens

        cardLayout = new CardLayout();

        mainPanel = new JPanel(cardLayout);

        // Create starting screen

        JPanel startScreen = createStartScreen();

        mainPanel.add(startScreen, "START");

        // Create end screen (will be updated later)

        endScreen = new JPanel();

        mainPanel.add(endScreen, "END");

        add(mainPanel);

        // Show start screen initially

        cardLayout.show(mainPanel, "START");

    }

    void parseCSV(File file) {

        yearCountMap.clear();

        try (BufferedReader br = new BufferedReader(new FileReader(file))) {

            String line = br.readLine(); // Skip header

            while ((line = br.readLine()) != null) {

                String[] tokens = line.split(",");

                if (tokens.length >= 2) {

                    String year = tokens[1].trim();

                    yearCountMap.put(year, yearCountMap.getOrDefault(year, 0) + 1);

                }

            }

        } catch (IOException e) {

            JOptionPane.showMessageDialog(this, "Error reading file: " + e.getMessage());

        }

        if (!yearCountMap.isEmpty()) {

            List<String> yearList = new ArrayList<>(yearCountMap.keySet());

            Collections.sort(yearList);

            years = yearList.toArray(new String[0]);

            placedStudents = new int[years.length];

            for (int i = 0; i < years.length; i++) {

                placedStudents[i] = yearCountMap.get(years[i]);

            }

        }

    }

    void trainNeuralNetwork() {

        if (placedStudents.length < 3) {

            JOptionPane.showMessageDialog(this, "Need at least 3 years of data to train!");

            return;

        }

        // Initialize and train neural network

        neuralNetwork = new NeuralNetwork(3, 10, 5, 1);

        neuralNetwork.train(inputs, outputs, 1000, 0.1);

    }

    boolean showYearWithDLPrediction(String inputYear) {

        if (neuralNetwork == null) {

            return false;

        }

        isDLPrediction = true;

        int inputYearNum;

        try {

            inputYearNum = Integer.parseInt(inputYear);

        } catch (NumberFormatException e) {

            return false;

        }

        if (yearCountMap.containsKey(inputYear)) {

            highlightedYear = inputYear;

            predictedPercentage = null;

            highlightedCount = yearCountMap.get(inputYear);

            repaint();

            return true;

        }

        if (years.length == 0)

            return false;

        int lastKnownYear = Integer.parseInt(years[years.length - 1]);

        if (inputYearNum <= lastKnownYear)

            return false;

        // Use neural network for prediction

        double maxCount = Arrays.stream(placedStudents).max().orElse(1);

        double[] input = new double[3];

        input[0] = (double) (inputYearNum - Integer.parseInt(years[0])) / 10.0;

        input[1] = placedStudents[placedStudents.length - 1] / maxCount;

        input[2] = (placedStudents[placedStudents.length - 1] - placedStudents[placedStudents.length - 2]) / maxCount;

        double[] prediction = neuralNetwork.predict(input);

        int predictedCount = (int) (prediction[0] \* maxCount);

        if (predictedCount < 0)

            predictedCount = 0;

        int lastYearCount = placedStudents[placedStudents.length - 1];

        if (lastYearCount != 0) {

            double rawPercentage = ((double) (predictedCount - lastYearCount) / lastYearCount) \* 100;

            if (rawPercentage > 100)

                rawPercentage = 100;

            else if (rawPercentage < -100)

                rawPercentage = -100;

            predictedPercentage = rawPercentage;

        } else {

            predictedPercentage = null;

        }

        highlightedYear = inputYear;

        highlightedCount = predictedCount;

        repaint();

        return true;

    }

    boolean hasData() {

        return !yearCountMap.isEmpty();

    }

    @Override

    protected void paintComponent(Graphics g) {

        super.paintComponent(g);

        Graphics2D g2 = (Graphics2D) g;

        int width = getWidth();

        int height = getHeight();

        int padding = 50;

        if (placedStudents == null || placedStudents.length == 0) {

            Font headingFont = new Font("SansSerif", Font.BOLD, 16);

            g2.setFont(headingFont);

            g2.setColor(Color.BLACK);

            String heading = "Placement trend prediction with Deep Learning for engineering college";

            g2.drawString(heading, 20, 40);

            Font normalFont = new Font("SansSerif", Font.PLAIN, 12);

            Font boldFont = new Font("SansSerif", Font.BOLD, 12);

            String[] lines = {

                    "This application uses deep learning to analyze placement data and predict future trends.",

                    "",

                    "Step To Use This Application :",

                    "1. Click 'Upload CSV' button.",

                    "2. Select your CSV file with 'Name,Year' data.",

                    "3. Click 'Train Model' to train the neural network.",

                    "4. Enter a year and click 'DL Prediction' for deep learning based prediction.",

                    "5. The graph shows prediction percentage for upcoming years.",

                    "",

                    "Created By",

                    "Rohan Ingle"

            };

            int x = 20;

            int y = 60;

            int lineHeight = g2.getFontMetrics().getHeight();

            for (String line : lines) {

                if (line.equals("Step To Use This Application :") || line.equals("Created By")) {

                    g2.setFont(boldFont);

                } else {

                    g2.setFont(normalFont);

                }

                g2.drawString(line, x, y);

                y += lineHeight;

            }

            return;

        }

        // Draw the graph when data is loaded

        g2.setColor(Color.BLACK);

        g2.drawLine(padding, padding, padding, height - padding);

        g2.drawLine(padding, height - padding, width - padding, height - padding);

        for (int i = 0; i < placedStudents.length; i++) {

            int x = padding + i \* (barWidth + 10) + 5;

            int barHeight = placedStudents[i] \* (height - 2 \* padding) / maxValue;

            int y = height - padding - barHeight;

            g2.setColor(Color.BLUE);

            g2.fillRect(x, y, barWidth, barHeight);

            g2.setColor(Color.BLACK);

            g2.drawString(years[i], x, height - padding + 15);

        }

            // Add DL indicator

                if (isDLPrediction) {

                    g2.setColor(Color.DARK\_GRAY);

                    g2.setFont(new Font("SansSerif", Font.BOLD, 10));

                    g2.drawString("DL", x + barWidth - 20, y - 5);

                }

            }

        }

    }

}

// Neural Network implementation in pure Java

class NeuralNetwork {

    private int inputSize;

    private int hiddenSize1;

    private int hiddenSize2;

    private int outputSize;

    private double[][] weightsInputHidden1;

    private double[][] weightsHidden1Hidden2;

    private double[][] weightsHidden2Output;

    private double[] biasHidden1;

    private double[] biasHidden2;

    private double[] biasOutput;

    private Random random = new Random();

    public NeuralNetwork(int inputSize, int hiddenSize1, int hiddenSize2, int outputSize) {

        this.inputSize = inputSize;

        this.hiddenSize1 = hiddenSize1;

        this.hiddenSize2 = hiddenSize2;

        this.outputSize = outputSize;

        // Initialize weights and biases

        weightsInputHidden1 = new double[inputSize][hiddenSize1];

        weightsHidden1Hidden2 = new double[hiddenSize1][hiddenSize2];

        weightsHidden2Output = new double[hiddenSize2][outputSize];

        biasHidden1 = new double[hiddenSize1];

        biasHidden2 = new double[hiddenSize2];

        biasOutput = new double[outputSize];

        initializeWeights();

    }

    private void initializeWeights() {

        // Xavier initialization

        double scale1 = Math.sqrt(2.0 / inputSize);

        double scale2 = Math.sqrt(2.0 / hiddenSize1);

        double scale3 = Math.sqrt(2.0 / hiddenSize2);

        for (int i = 0; i < inputSize; i++) {

            for (int j = 0; j < hiddenSize1; j++) {

                weightsInputHidden1[i][j] = random.nextGaussian() \* scale1;

            }

        }

        for (int i = 0; i < hiddenSize1; i++) {

            for (int j = 0; j < hiddenSize2; j++) {

                weightsHidden1Hidden2[i][j] = random.nextGaussian() \* scale2;

            }

        }

        for (int i = 0; i < hiddenSize2; i++) {

            for (int j = 0; j < outputSize; j++) {

                weightsHidden2Output[i][j] = random.nextGaussian() \* scale3;

            }

        }

        for (int i = 0; i < hiddenSize1; i++) {

            biasHidden1[i] = 0.01;

        }

        for (int i = 0; i < hiddenSize2; i++) {

            biasHidden2[i] = 0.01;

        }

        for (int i = 0; i < outputSize; i++) {

            biasOutput[i] = 0.01;

        }

    }

    private double relu(double x) {

        return Math.max(0, x);

    }

    private double reluDerivative(double x) {

        return x > 0 ? 1 : 0;

    }

    private double sigmoid(double x) {

        return 1.0 / (1.0 + Math.exp(-x));

    }

    public double[] predict(double[] input) {

        // Forward propagation

        double[] hidden1 = new double[hiddenSize1];

        double[] hidden2 = new double[hiddenSize2];

        double[] output = new double[outputSize];

        // Input to Hidden1

        for (int j = 0; j < hiddenSize1; j++) {

            double sum = biasHidden1[j];

            for (int i = 0; i < inputSize; i++) {

                sum += input[i] \* weightsInputHidden1[i][j];

            }

            hidden1[j] = relu(sum);

        }

        // Hidden1 to Hidden2

        for (int j = 0; j < hiddenSize2; j++) {

            double sum = biasHidden2[j];

            for (int i = 0; i < hiddenSize1; i++) {

                sum += hidden1[i] \* weightsHidden1Hidden2[i][j];

            }

            hidden2[j] = relu(sum);

        }

        // Hidden2 to Output

        for (int j = 0; j < outputSize; j++) {

            double sum = biasOutput[j];

            for (int i = 0; i < hiddenSize2; i++) {

                sum += hidden2[i] \* weightsHidden2Output[i][j];

            }

            output[j] = sigmoid(sum);

        }

        return output;

    }

    public void train(List<double[]> inputs, List<double[]> outputs, int epochs, double learningRate) {

        for (int epoch = 0; epoch < epochs; epoch++) {

            double totalLoss = 0;

            for (int sample = 0; sample < inputs.size(); sample++) {

                double[] input = inputs.get(sample);

                double[] target = outputs.get(sample);

                // Forward propagation

                double[] hidden1 = new double[hiddenSize1];

                double[] hidden1Raw = new double[hiddenSize1];

                double[] hidden2 = new double[hiddenSize2];

                double[] hidden2Raw = new double[hiddenSize2];

                double[] output = new double[outputSize];

                // Input to Hidden1

                for (int j = 0; j < hiddenSize1; j++) {

                    double sum = biasHidden1[j];

                    for (int i = 0; i < inputSize; i++) {

                        sum += input[i] \* weightsInputHidden1[i][j];

                    }

                    hidden1Raw[j] = sum;

                    hidden1[j] = relu(sum);

                }

                // Hidden1 to Hidden2

                for (int j = 0; j < hiddenSize2; j++) {

                    double sum = biasHidden2[j];

                    for (int i = 0; i < hiddenSize1; i++) {

                        sum += hidden1[i] \* weightsHidden1Hidden2[i][j];

                    }

                    hidden2Raw[j] = sum;

                    hidden2[j] = relu(sum);

                }

                // Hidden2 to Output

                for (int j = 0; j < outputSize; j++) {

                    double sum = biasOutput[j];

                    for (int i = 0; i < hiddenSize2; i++) {

                        sum += hidden2[i] \* weightsHidden2Output[i][j];

                    }

                    output[j] = sigmoid(sum);

                }

                // Calculate loss

                for (int i = 0; i < outputSize; i++) {

                    totalLoss += Math.pow(target[i] - output[i], 2);

                }

                // Backpropagation

                double[] outputError = new double[outputSize];

                double[] hidden2Error = new double[hiddenSize2];

                double[] hidden1Error = new double[hiddenSize1];

                // Output layer error

                for (int i = 0; i < outputSize; i++) {

                    outputError[i] = (output[i] - target[i]) \* output[i] \* (1 - output[i]);

                }

                // Hidden2 layer error

                for (int i = 0; i < hiddenSize2; i++) {

                    double error = 0;

                    for (int j = 0; j < outputSize; j++) {

                        error += outputError[j] \* weightsHidden2Output[i][j];

                    }

                    hidden2Error[i] = error \* reluDerivative(hidden2Raw[i]);

                }

                // Hidden1 layer error

                for (int i = 0; i < hiddenSize1; i++) {

                    double error = 0;

                    for (int j = 0; j < hiddenSize2; j++) {

                        error += hidden2Error[j] \* weightsHidden1Hidden2[i][j];

                    }

                    hidden1Error[i] = error \* reluDerivative(hidden1Raw[i]);

                }

                // Update weights and biases

                // Hidden2 to Output

                for (int i = 0; i < hiddenSize2; i++) {

                    for (int j = 0; j < outputSize; j++) {

                        weightsHidden2Output[i][j] -= learningRate \* outputError[j] \* hidden2[i];

                    }

                }

                for (int i = 0; i < outputSize; i++) {

                    biasOutput[i] -= learningRate \* outputError[i];

                }

                // Hidden1 to Hidden2

                for (int i = 0; i < hiddenSize1; i++) {

                    for (int j = 0; j < hiddenSize2; j++) {

                        weightsHidden1Hidden2[i][j] -= learningRate \* hidden2Error[j] \* hidden1[i];

                    }

                }

                for (int i = 0; i < hiddenSize2; i++) {

                    biasHidden2[i] -= learningRate \* hidden2Error[i];

                }

                // Input to Hidden1

                for (int i = 0; i < inputSize; i++) {

                    for (int j = 0; j < hiddenSize1; j++) {

                        weightsInputHidden1[i][j] -= learningRate \* hidden1Error[j] \* input[i];

                    }

                }

                for (int i = 0; i < hiddenSize1; i++) {

                    biasHidden1[i] -= learningRate \* hidden1Error[i];

                }

            }

            // Print progress every 100 epochs

            if (epoch % 100 == 0) {

                System.out.println("Epoch " + epoch + ", Loss: " + totalLoss / inputs.size());

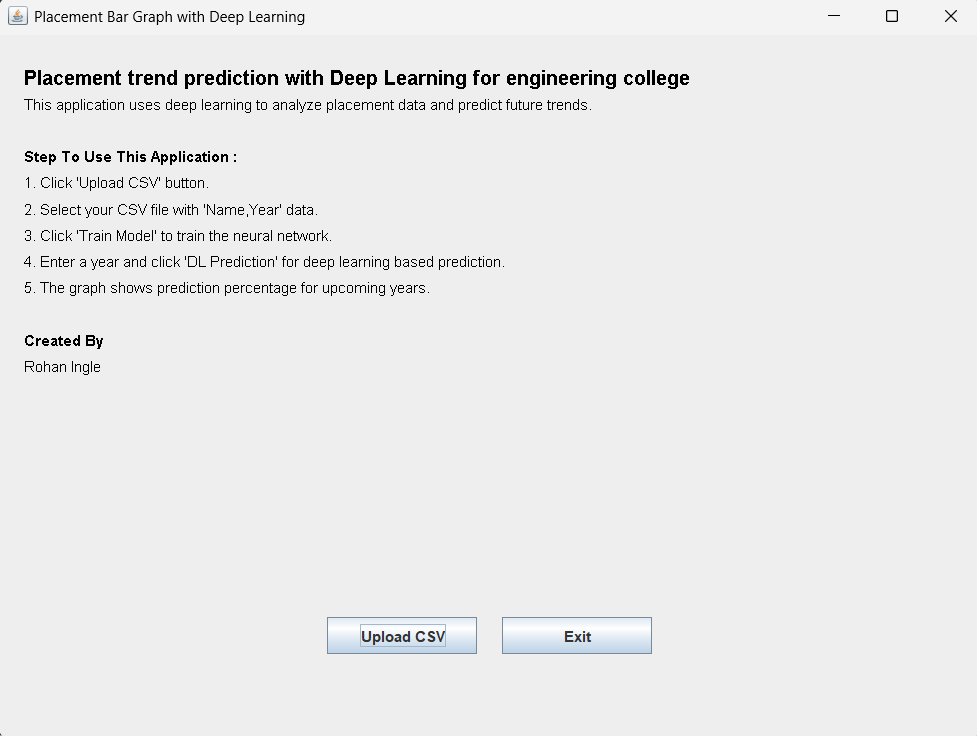
            }

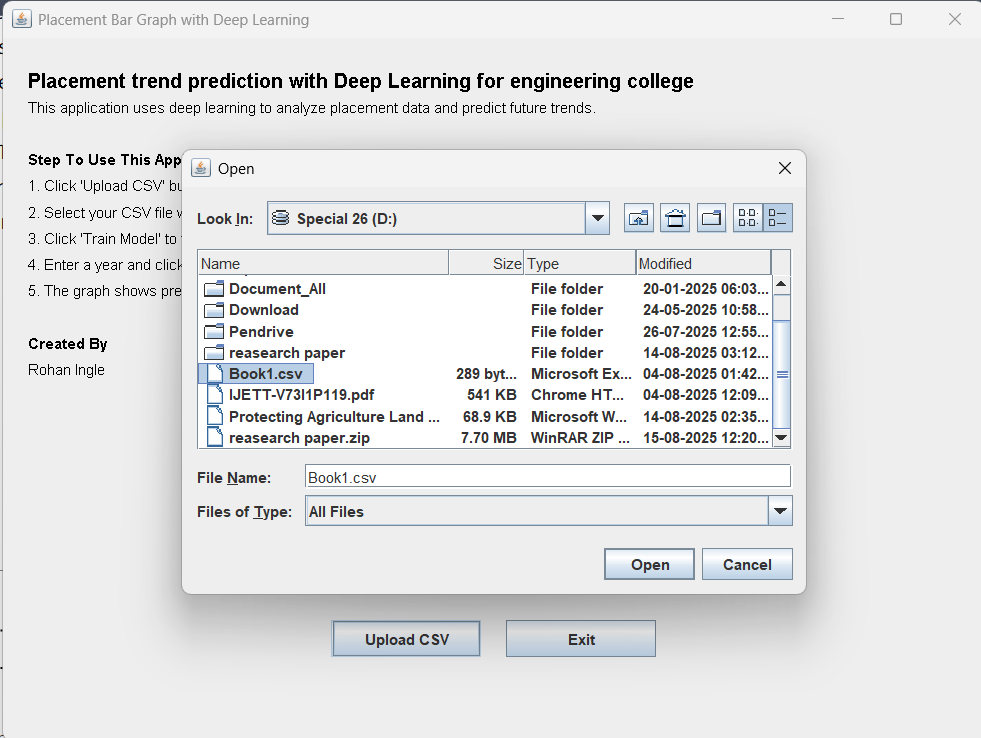
        }

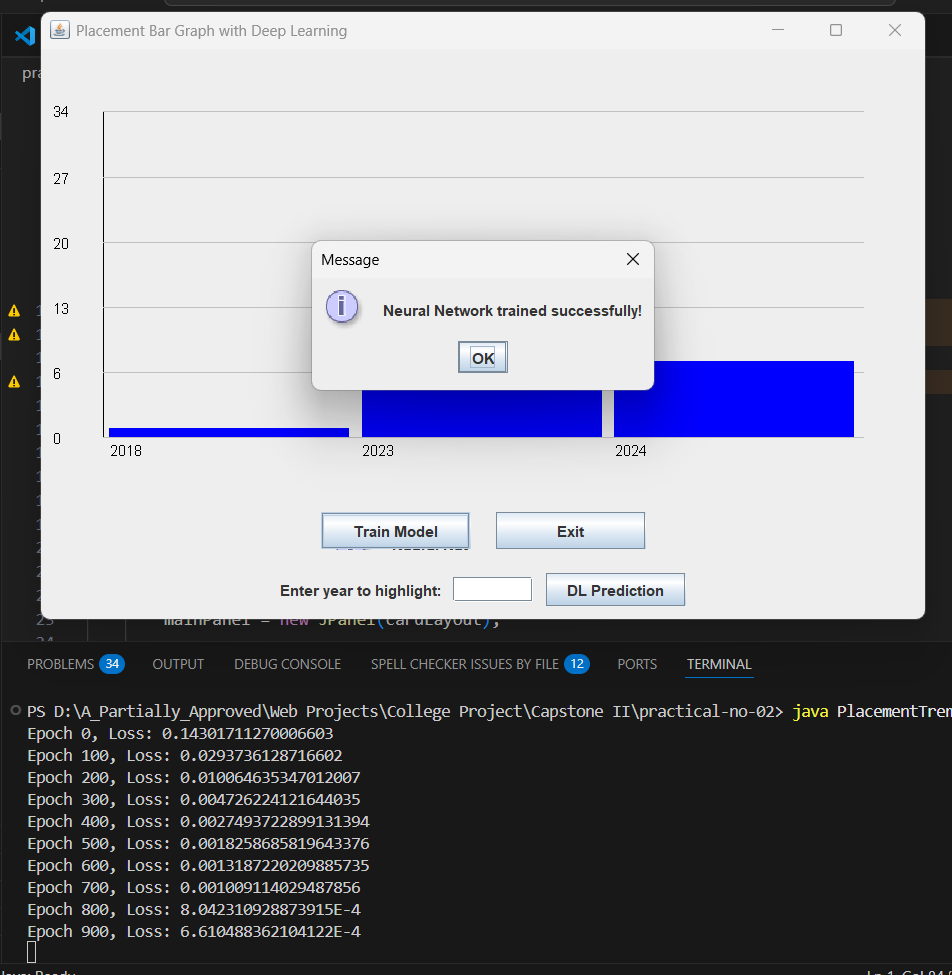
    }

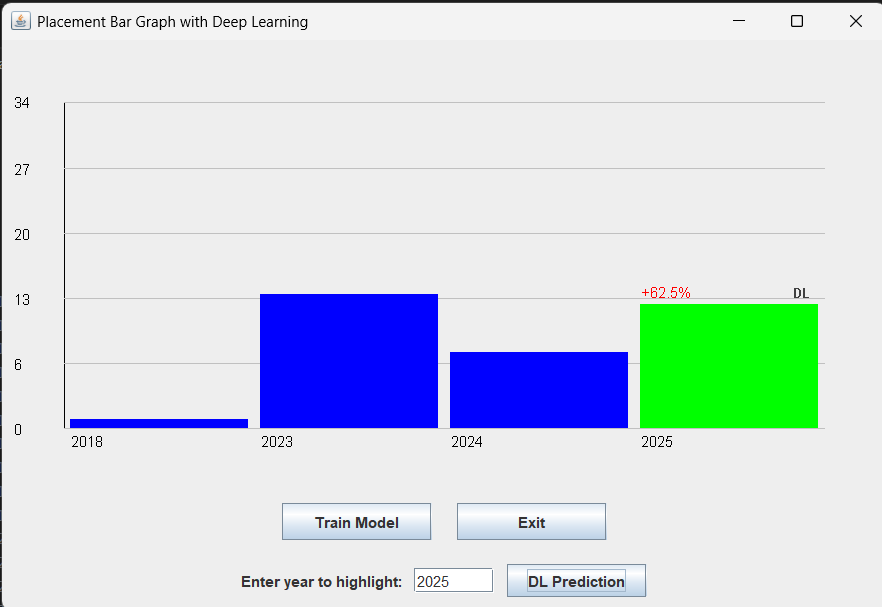
}

**Program Output :**

****

****

****

****

**Advantages** **:**

1. Predicts future placement trends using deep learning.
2. Visual bar graph makes analysis intuitive and impactful.
3. Flexible and extensible for updated data and model retraining.
4. Runs locally, preserving data confidentiality.
5. Can be customized for various datasets.

**Disadvantages :**

1. Considers only basic features (year, count, trend); external factors not included.
2. Prediction quality depends on data volume and accuracy.
3. No integration with external college management systems.
4. Only basic analytics and visualizations are available.

**Conclusion** **:**

The "Placement Training Prediction and Visual Analytics" platform for an engineering school effectively illustrates the combination of data-driven predictive analytics and visual analytics. Through the examination of student information, including academic achievement, skill sets, and participation in training, the platform forecasts possible placement results with high accuracy. The visual analytics module offers a transparent, interactive visualization of trends, facilitating students and administrators in decision-making processes for training interventions.