Heart Disease Prediction Using Neural Networks

Case Study Report

---

1. Problem Description and Rationale

Problem Statement

Heart disease is one of the leading causes of death worldwide. Early detection can significantly improve patient outcomes through timely treatment. This project predicts whether a patient has heart disease based on 13 clinical features.

Why Neural Networks?

- **Non-linear relationships**: Heart disease risk factors interact in complex ways
- **Automatic feature learning**: No need for manual feature engineering
- **Probabilistic output**: Provides confidence scores for predictions

---

2. Dataset Description and Preprocessing

Dataset Overview

- **Source**: Cleveland Heart Disease Dataset (UCI Repository)
- **Size**: 303 patients
- **Features**: 13 clinical attributes
- **Target**: Binary (0 = No disease, 1 = Disease)

Features

Feature	Description
age	Age in years
sex	Gender (0=F, 1=M)
cp	Chest pain type (1-4)
trestbps	Resting blood pressure
chol	Serum cholesterol

Feature	Description
fbs	Fasting blood sugar > 120
restecg	Resting ECG results
thalach	Max heart rate achieved
exang	Exercise induced angina
oldpeak	ST depression
slope	Slope of ST segment
ca	Number of vessels colored
thal	Thalassemia type

Preprocessing Steps

1. Normalization

```
Model.normalizeMinMax(X);
```

Applied Min-Max scaling to normalize all features to [0, 1] range. This ensures:

- All features contribute equally to learning
- Faster convergence during training
- Better performance with activation functions

2. Train-Test Split

```
SplitResult split = Model.trainTestSplit(X, Y, 0.3);
```

- Training: 70% (212 samples)
- Testing: 30% (91 samples)

---

3. Neural Network Architecture

Network Structure

Architecture: 13 → 32 → 16 → 1

```
model.createNetwork(3,
    new int[] { 13, 32, 16 },
```

new int[] { 32, 16, 1 });

Input Layer: 13 neurons (matches 13 features)

Hidden Layer 1: 32 neurons + ReLU

Hidden Layer 2: 16 neurons + ReLU

Output Layer: 1 neuron + Sigmoid

Why This Architecture?

Layer Sizes:

- **13 → 32**: First layer expands to capture feature combinations and interactions
- **32 → 16**: Second layer compresses information, learning important patterns
- **16 → 1**: Final layer produces single probability output

This creates a bottleneck structure that forces the network to learn the most important features.

Activation Functions

Hidden Layers: ReLU

model.setActivationFunction(new ReLU());

Why ReLU for hidden layers?

- **Fast training**: Simple computation (max(0, x))
- **No vanishing gradients**: Gradient is either 0 or 1
- **Sparse activation**: Some neurons turn off, making the network efficient
- **Standard choice**: Works well for most classification tasks

Output Layer: Sigmoid

model.getNetwork().getLayers()[2].setActivationFunction(new Sigmoid());

Why Sigmoid for output?

- **Probability output**: Produces values between 0 and 1
- **Binary classification**: Perfect for yes/no decisions (disease or no disease)
- **Interpretable**: Output can be read as "confidence" percentage

---

4. Hyperparameter Configuration

```
model.setWeightInitializer(new XavierNormal());
model.setLossFunction(new BinaryCrossEntropy());
model.setActivationFunction(new ReLU());
model.setLearningRate(0.003);
model.setBatchSize(32);
model.setEpoches(5000);
```

Parameter	Value	Why?
Learning Rate	0.003	Balanced speed - not too fast (unstable) or slow (takes forever)
Batch Size	32	Gives ~7 batches per epoch for smooth learning
Epochs	5000	Enough iterations to fully converge
Weight Init	Xavier Normal	Keeps activations stable across layers
Loss Function	Binary Cross-Entropy	Standard for binary classification, better gradients than MSE

Why Xavier Initialization?

Initializes weights based on layer sizes: W ~ N(0, sqrt(2/(input_size + output_size)))

- Prevents vanishing/exploding gradients
- Helps network learn from the start

Why Binary Cross-Entropy Loss?

Formula: L = -[y*log(ŷ) + (1-y)*log(1-ŷ)]

- Strongly penalizes wrong confident predictions
- Perfect match for Sigmoid output (both deal with probabilities)
- Better than MSE for classification

---

5. Training and Results

Training Process

- **Training samples**: 212 patients
- **Test samples**: 91 patients
- **Total epochs**: 5000
- **Gradient updates per epoch**: ~7 (212/32)

Performance Metrics (from one of the trained models)

Metric	Value
Accuracy (total correct)	90.00%
Precision (correct when predicting disease)	88.10%
Recall (% of actual disease cases caught)	90.24%

Confusion Matrix

	Predicted No	Predicted Yes
Actual No	40	5
Actual Yes	4	42

What this means:

- **40 True Negatives**: Correctly identified healthy patients
- **42 True Positives**: Correctly identified disease patients
- **5 False Positives**: Healthy patients flagged as sick (safe error)
- **4 False Negatives**: Disease patients missed (critical error)

Note: we made that predictions that are equal or higher than 0.5 are equal to 1 or equal to that person is having a heart disease and predictions that are less than 0.5 are equal to 0 or equal to that person is not having a heart disease

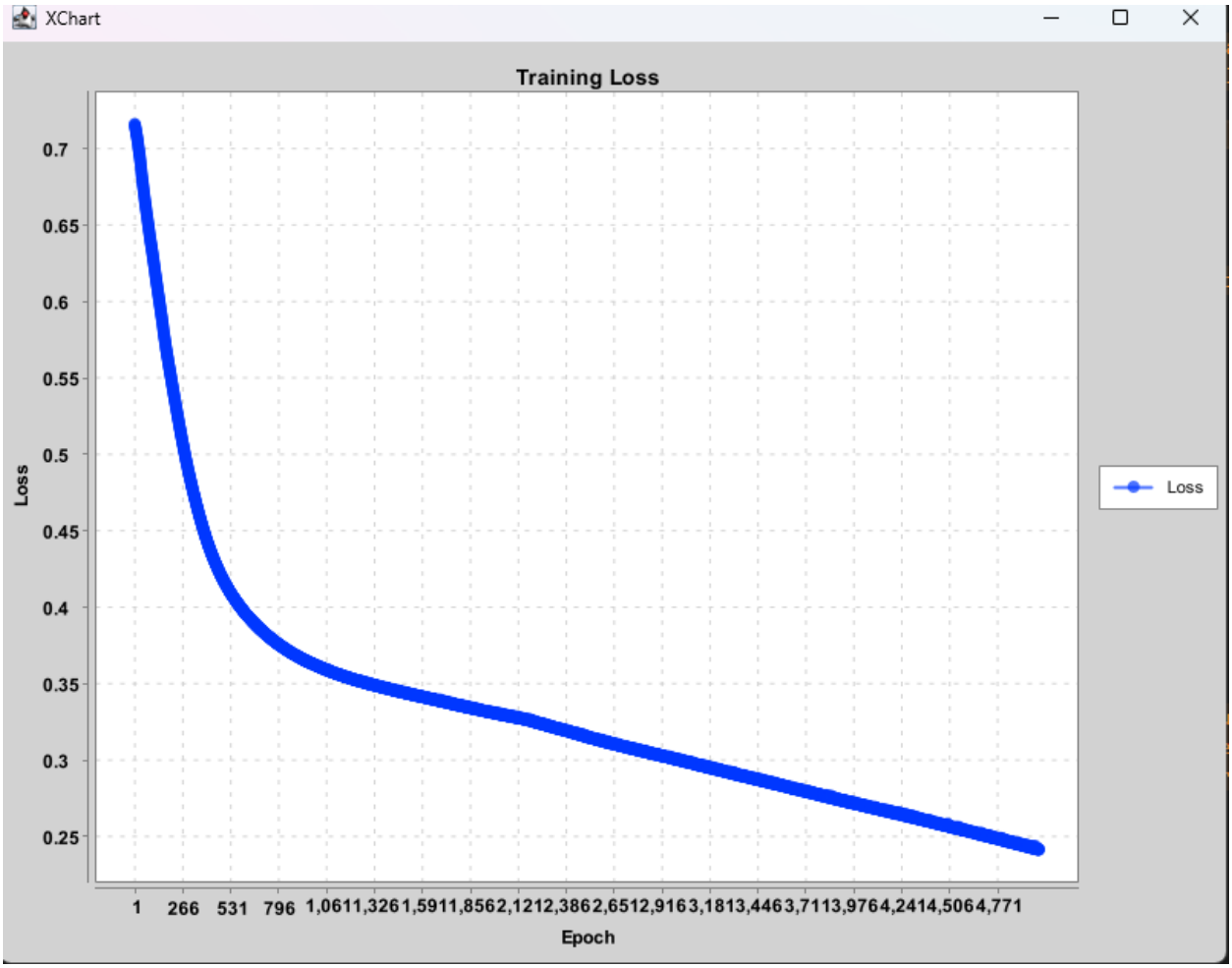
Loss Curve

-Rapid learning: Loss drops sharply from 0.72 to 0.40 in first 500 epochs**

-Smooth convergence: Gradual decrease to ~0.25 with no oscillations**

****Stable training**: No signs of overfitting or instability**

****Optimal stopping**: Plateau around epoch 3000 indicates sufficient training**



---

6. How the Library Was Used

Step 1: Load and Preprocess Data

(CSVHelper is not part of the library)

```
double[][] X = CSVHelper.selectColumns("Heart_disease_cleveland_new.csv",
    new int[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 }, 1);
double[][] Y = CSVHelper.selectColumns("Heart_disease_cleveland_new.csv",
    new int[] { 13 }, 1);
```

(the number besides the column numbers is how many rows we want to skip its useful for skipping feature names)

```
Model.normalizeMinMax(X);
SplitResult split = Model.trainTestSplit(X, Y, 0.3);
```

Step 2: Configure Model

```
Model model = new Model();
model.setWeightInitializer(new XavierNormal());
model.setLossFunction(new BinaryCrossEntropy());
model.setActivationFunction(new ReLU());
model.setLearningRate(0.003);
model.setBatchSize(32);
model.setEpoches(5000);
```

Step 3: Build Network

```
model.createNetwork(3,
    new int[] { 13, 32, 16 },
    new int[] { 32, 16, 1 });

model.getNetwork().getLayers()[2].setActivationFunction(new Sigmoid());
```

Step 4: Train

```
model.train(split.X_train, split.Y_train);
```

Step 5: Evaluate

```
double[][] predictions = model.predict(split.X_test);

// Calculate metrics
for (int i = 0; i < predictions.length; i++) {
    int pred = predictions[i][0] >= 0.5 ? 1 : 0;
    int actual = (int) split.Y_test[i][0];
    // Count TP, TN, FP, FN...
}
```

Step 6: Save Model and test data

```
CSVHelper.writeCSV("X_test.csv", split.X_test);
```

CSVHelper.writeCSV("Y_test.csv", split.Y_test);

model.saveModel("model.nn");

If we want to use the model directly without retraining it we can just use Model model = Model.loadModel("model.nn"); and load the test data back (double[][] X_test = CSVHelper.selectColumns("X_test.csv",

new int[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 }, 0);

double[][] Y_test = CSVHelper.selectColumns("Y_test.csv", new int[] { 0 }, 0);

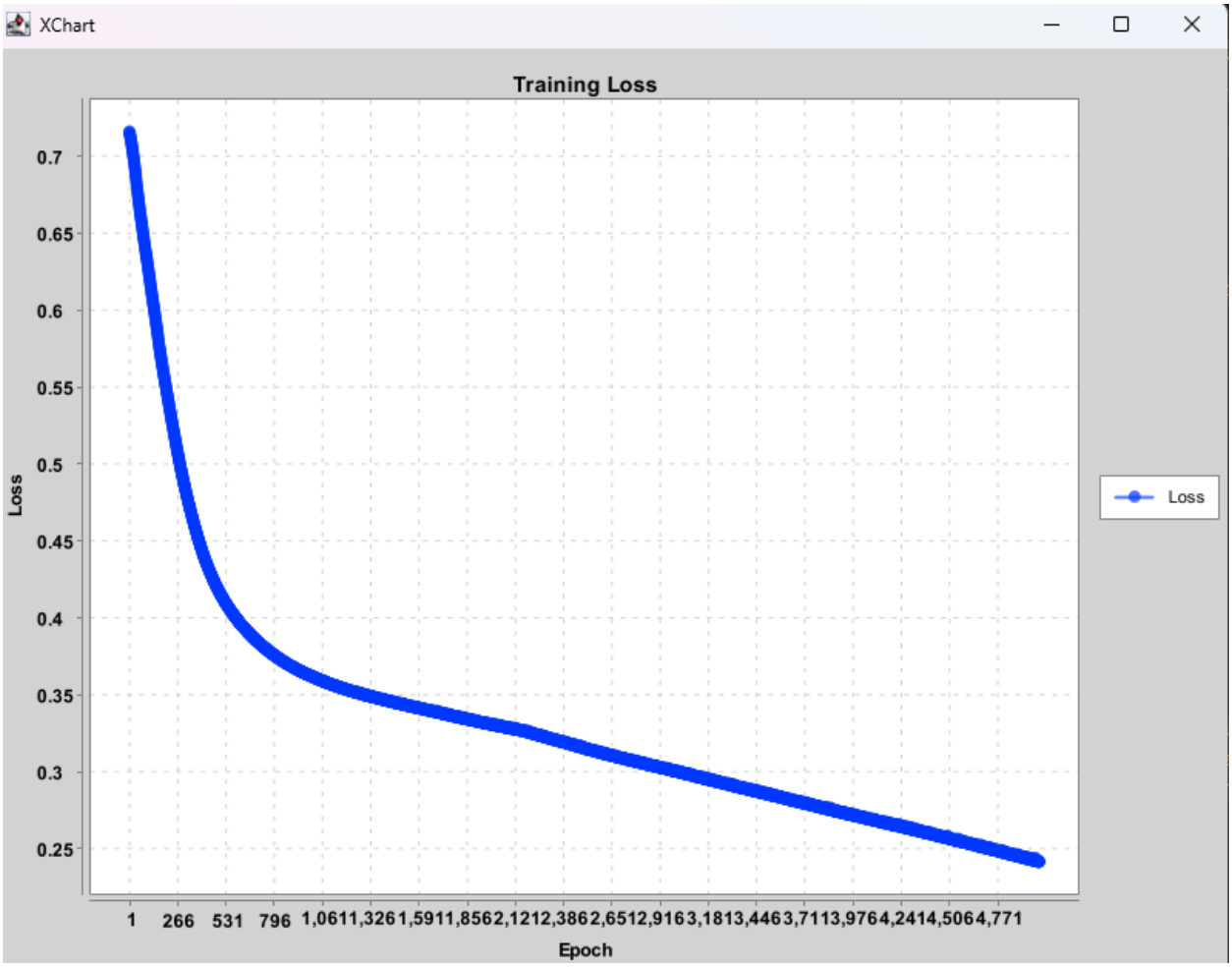
)

(we didn’t skip the feature names this time because it does not exist in the test data csv file)

---

7. Results Screenshots

Loss curve graph picture



Results console output:

```
Accuracy (total correct): 90.0%
Precision (correct when predicting disease): 88.09523809523809%
Recall (% of actual disease cases caught): 90.2439024390244%
```