Code:  
#include <iostream>

#include <fstream>

#include <sstream>

#include <vector>

#include <map>

#include <cmath>

#include <cstdlib>

#include <ctime>

using namespace std;

/\*\*

\* @struct Example

\* @brief Stores each example's encoded features and label.

\*

\* This structure represents each individual data point in the dataset.

\* - `features`: A vector that holds the encoded values for each categorical feature.

\* - `label`: A binary outcome representing the user action (0 = "skips", 1 = "reads").

\*

\* For example, a user who is "unknown", views a "new" thread of "long" length at "work"

\* could be represented as:

\* - features: {0.0, 1.0, 1.0, 0.0} (encoded values for categorical features)

\* - label: 1 (assume "reads")

\*/

struct Example {

vector<double> features; // Encoded values for input features

int label; // 0 for "skips" and 1 for "reads"

};

/\*\*

\* @class NeuralNetwork

\* @brief Implements a neural network with a single hidden layer for binary classification.

\*

\* This class represents a simple feed-forward neural network with:

\* - An input layer (4 features)

\* - A single hidden layer (2 units)

\* - An output layer (1 output for binary classification)

\*

\* The network uses sigmoid activation functions and trains with gradient descent.

\*/

class NeuralNetwork {

private:

// Member Variables

double weightsInputHidden[4][2]; // Weights from input to hidden layer

double weightsHiddenOutput[2]; // Weights from hidden to output layer

double hiddenBias[2], outputBias; // Biases for hidden and output layers

double learningRate; // Step size for weight updates

/\*\*

\* @brief Sigmoid activation function

\*

\* @param x Input value

\* @return Sigmoid of x, a value between 0 and 1

\*

\* Example:

\* If `x` = 0, then `sigmoid(0)` = 0.5.

\*/

double sigmoid(double x) {

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

}

public:

/\*\*

\* @brief Constructor to initialize weights and biases.

\*

\* Initializes weights and biases with small random values for symmetry breaking.

\* Uses a fixed seed for reproducibility and sets the learning rate.

\*

\* @param lr Learning rate for training (e.g., 0.1)

\*/

NeuralNetwork(double lr) : learningRate(lr) {

srand(42); // Fixed seed for reproducibility

for (int i = 0; i < 4; ++i)

for (int j = 0; j < 2; ++j)

weightsInputHidden[i][j] = (rand() / double(RAND\_MAX)) \* 0.1 - 0.05;

for (int j = 0; j < 2; ++j)

weightsHiddenOutput[j] = (rand() / double(RAND\_MAX)) \* 0.1 - 0.05;

hiddenBias[0] = (rand() / double(RAND\_MAX)) \* 0.1 - 0.05;

hiddenBias[1] = (rand() / double(RAND\_MAX)) \* 0.1 - 0.05;

outputBias = (rand() / double(RAND\_MAX)) \* 0.1 - 0.05;

}

/\*\*

\* @brief Forward propagation through the network

\*

\* Computes the activations for the hidden layer and output layer.

\*

\* @param example The input example with features

\* @param hiddenLayer Array to store hidden layer activations

\* @return Output activation (prediction) for the given example

\*

\* Example:

\* Suppose `example.features` is {0.0, 1.0, 1.0, 0.0}, and initial weights and biases

\* are small random values. This function computes hidden layer activations and a final

\* output activation (e.g., 0.72), which can be interpreted as the probability of "reads".

\*/

double forward(const Example& example, double hiddenLayer[2]) {

// Compute activations for hidden layer neurons

for (int i = 0; i < 2; ++i) {

hiddenLayer[i] = hiddenBias[i];

for (int j = 0; j < 4; ++j) {

hiddenLayer[i] += example.features[j] \* weightsInputHidden[j][i];

}

hiddenLayer[i] = sigmoid(hiddenLayer[i]); // Activation for hidden neuron

}

// Compute activation for output layer neuron

double output = outputBias;

for (int i = 0; i < 2; ++i) {

output += hiddenLayer[i] \* weightsHiddenOutput[i];

}

return sigmoid(output); // Final output prediction

}

/\*\*

\* @brief Backward propagation for weight updates

\*

\* Computes gradients for output and hidden layers, then updates weights and biases

\* using the gradient descent algorithm.

\*

\* @param example The input example (features and label)

\* @param output The computed output from forward propagation

\* @param hiddenLayer Hidden layer activations

\*

\* Example:

\* Given `output` = 0.72 and `example.label` = 1, this function calculates the error and

\* adjusts weights and biases to reduce the error in subsequent iterations.

\*/

void backward(const Example& example, double output, double hiddenLayer[2]) {

// Calculate error and delta for output layer

double outputError = example.label - output;

double outputDelta = outputError \* output \* (1 - output); // Gradient for output

// Calculate errors and deltas for hidden layer

double hiddenError[2];

double hiddenDelta[2];

for (int i = 0; i < 2; ++i) {

hiddenError[i] = outputDelta \* weightsHiddenOutput[i];

hiddenDelta[i] = hiddenError[i] \* hiddenLayer[i] \* (1 - hiddenLayer[i]);

}

// Update weights and biases for output layer

for (int i = 0; i < 2; ++i) {

weightsHiddenOutput[i] += learningRate \* outputDelta \* hiddenLayer[i];

}

outputBias += learningRate \* outputDelta;

// Update weights and biases for hidden layer

for (int i = 0; i < 2; ++i) {

for (int j = 0; j < 4; ++j) {

weightsInputHidden[j][i] += learningRate \* hiddenDelta[i] \* example.features[j];

}

hiddenBias[i] += learningRate \* hiddenDelta[i];

}

}

/\*\*

\* @brief Train the neural network on a given dataset

\*

\* Runs forward and backward propagation for each example in the dataset over multiple epochs.

\* Reports misclassified examples after each epoch for monitoring learning progress.

\*

\* @param train\_data Training dataset of examples

\* @param epochs Number of training epochs (e.g., 80)

\*/

void train(vector<Example>& train\_data, int epochs) {

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

int misclassified = 0;

for (auto& example : train\_data) {

double hiddenLayer[2];

double output = forward(example, hiddenLayer);

backward(example, output, hiddenLayer);

// Count misclassifications

if ((output >= 0.5) != example.label) {

misclassified++;

}

}

cout << "Epoch " << epoch + 1 << ": Misclassified = " << misclassified << "\n";

}

}

/\*\*

\* @brief Test the neural network on a test dataset

\*

\* Computes the number of misclassified examples for evaluating the network's performance.

\*

\* @param test\_data Testing dataset of examples

\* @return Number of misclassified examples in the test dataset

\*/

int test(const vector<Example>& test\_data) {

int misclassified = 0;

for (const auto& example : test\_data) {

double hiddenLayer[2];

double output = forward(example, hiddenLayer);

if ((output >= 0.5) != example.label) {

misclassified++;

}

}

return misclassified;

}

/\*\*

\* @brief Calculate accuracy on the test dataset

\*

\* Computes accuracy as the percentage of correctly classified examples.

\*

\* @param test\_data Testing dataset of examples

\* @return Accuracy as a percentage of correct predictions

\*/

double calculate\_accuracy(const vector<Example>& test\_data) {

int correct = 0;

for (const auto& example : test\_data) {

double hiddenLayer[2];

double output = forward(example, hiddenLayer);

if ((output >= 0.5) == example.label) {

correct++;

}

}

return static\_cast<double>(correct) / test\_data.size();

}

};

/\*\*

\* @brief Load data from a CSV file and encode categorical features.

\*

\* Reads CSV data, converts categorical features to numerical values, and creates

\* examples with features and labels.

\*

\* @param filename Path to the CSV file (e.g., "dataset.csv")

\* @return Encoded dataset

\*/

vector<Example> load\_data(const string &filename) {

vector<Example> data;

ifstream file(filename);

string line;

// Skip header line

getline(file, line);

// Load data from file and encode each example

while (getline(file, line)) {

stringstream ss(line);

Example example;

string word;

// Encode each feature as a numeric value (1.0 for specific values, 0.0 otherwise)

getline(ss, word, ','); // Author

example.features.push\_back(word == "known" ? 1.0 : 0.0);

getline(ss, word, ','); // Thread

example.features.push\_back(word == "new" ? 1.0 : 0.0);

getline(ss, word, ','); // Length

example.features.push\_back(word == "long" ? 1.0 : 0.0);

getline(ss, word, ','); // Where\_read

example.features.push\_back(word == "home" ? 1.0 : 0.0);

getline(ss, word, ','); // User\_action

example.label = (word == "reads") ? 1 : 0;

data.push\_back(example);

}

return data;

}

/\*\*

\* @brief Split dataset into training and testing sets based on a specified ratio.

\*

\* Divides the dataset into two parts based on a given training ratio (e.g., 0.8 for 80% train).

\*

\* @param data Full dataset

\* @param train Vector to store training set examples

\* @param test Vector to store testing set examples

\* @param train\_ratio Proportion of examples for training (0.8 means 80% train, 20% test)

\*/

void split\_data(const vector<Example>& data, vector<Example>& train, vector<Example>& test, float train\_ratio) {

int train\_size = static\_cast<int>(data.size() \* train\_ratio);

train.assign(data.begin(), data.begin() + train\_size);

test.assign(data.begin() + train\_size, data.end());

}

int main() {

string filename = "dataset.csv"; // Dataset file path

vector<Example> data = load\_data(filename);

vector<float> ratios = {0.8, 0.7, 0.5}; // Different train-test split ratios

for (float ratio : ratios) {

vector<Example> train\_data, test\_data;

split\_data(data, train\_data, test\_data, ratio);

NeuralNetwork nn(0.7); // Initialize network with a learning rate of 0.7

nn.train(train\_data, 80); // Train the network for 80 epochs

int testMisclassified = nn.test(test\_data);

double accuracy = nn.calculate\_accuracy(test\_data);

cout << "Test Misclassified = " << testMisclassified << "\n";

cout << "Accuracy for train:test ratio " << ratio << ":" << (1 - ratio) << " is " << accuracy << endl;

// Predictions for specific examples (e19 and e20)

Example e19 = {{0.0, 1.0, 1.0, 0.0}, -1}; // Encoded for "unknown", "new", "long", "work"

Example e20 = {{0.0, 0.0, 0.0, 1.0}, -1}; // Encoded for "unknown", "followup", "short", "home"

double hiddenLayer[2];

cout << "Prediction for e19: " << (nn.forward(e19, hiddenLayer) >= 0.5 ? "reads" : "skips") << endl;

cout << "Prediction for e20: " << (nn.forward(e20, hiddenLayer) >= 0.5 ? "reads" : "skips") << endl;

}

return 0;

}

Output:  
Epoch 1: Misclassified = 5

Epoch 2: Misclassified = 5

Epoch 3: Misclassified = 5

Epoch 4: Misclassified = 5

Epoch 5: Misclassified = 5

Epoch 6: Misclassified = 5

Epoch 7: Misclassified = 5

Epoch 8: Misclassified = 5

Epoch 9: Misclassified = 5

Epoch 10: Misclassified = 5

Epoch 11: Misclassified = 5

Epoch 12: Misclassified = 5

Epoch 13: Misclassified = 5

Epoch 14: Misclassified = 5

Epoch 15: Misclassified = 5

Epoch 16: Misclassified = 5

Epoch 17: Misclassified = 5

Epoch 18: Misclassified = 5

Epoch 19: Misclassified = 5

Epoch 20: Misclassified = 5

Epoch 21: Misclassified = 5

Epoch 22: Misclassified = 5

Epoch 23: Misclassified = 5

Epoch 24: Misclassified = 5

Epoch 25: Misclassified = 5

Epoch 26: Misclassified = 5

Epoch 27: Misclassified = 5

Epoch 28: Misclassified = 5

Epoch 29: Misclassified = 5

Epoch 30: Misclassified = 5

Epoch 31: Misclassified = 3

Epoch 32: Misclassified = 3

Epoch 33: Misclassified = 2

Epoch 34: Misclassified = 1

Epoch 35: Misclassified = 1

Epoch 36: Misclassified = 1

Epoch 37: Misclassified = 1

Epoch 38: Misclassified = 1

Epoch 39: Misclassified = 1

Epoch 40: Misclassified = 1

Epoch 41: Misclassified = 1

Epoch 42: Misclassified = 1

Epoch 43: Misclassified = 1

Epoch 44: Misclassified = 1

Epoch 45: Misclassified = 1

Epoch 46: Misclassified = 1

Epoch 47: Misclassified = 1

Epoch 48: Misclassified = 1

Epoch 49: Misclassified = 1

Epoch 50: Misclassified = 1

Epoch 51: Misclassified = 1

Epoch 52: Misclassified = 1

Epoch 53: Misclassified = 1

Epoch 54: Misclassified = 0

Epoch 55: Misclassified = 0

Epoch 56: Misclassified = 0

Epoch 57: Misclassified = 0

Epoch 58: Misclassified = 0

Epoch 59: Misclassified = 0

Epoch 60: Misclassified = 0

Epoch 61: Misclassified = 0

Epoch 62: Misclassified = 0

Epoch 63: Misclassified = 0

Epoch 64: Misclassified = 0

Epoch 65: Misclassified = 0

Epoch 66: Misclassified = 0

Epoch 67: Misclassified = 0

Epoch 68: Misclassified = 0

Epoch 69: Misclassified = 0

Epoch 70: Misclassified = 0

Epoch 71: Misclassified = 0

Epoch 72: Misclassified = 0

Epoch 73: Misclassified = 0

Epoch 74: Misclassified = 0

Epoch 75: Misclassified = 0

Epoch 76: Misclassified = 0

Epoch 77: Misclassified = 0

Epoch 78: Misclassified = 0

Epoch 79: Misclassified = 0

Epoch 80: Misclassified = 0

Test Misclassified = 0

Accuracy for train:test ratio 0.8:0.2 is 1

Prediction for e19: skips

Prediction for e20: skips

Epoch 1: Misclassified = 3

Epoch 2: Misclassified = 3

Epoch 3: Misclassified = 3

Epoch 4: Misclassified = 3

Epoch 5: Misclassified = 3

Epoch 6: Misclassified = 3

Epoch 7: Misclassified = 3

Epoch 8: Misclassified = 3

Epoch 9: Misclassified = 3

Epoch 10: Misclassified = 3

Epoch 11: Misclassified = 3

Epoch 12: Misclassified = 3

Epoch 13: Misclassified = 3

Epoch 14: Misclassified = 3

Epoch 15: Misclassified = 3

Epoch 16: Misclassified = 3

Epoch 17: Misclassified = 3

Epoch 18: Misclassified = 3

Epoch 19: Misclassified = 3

Epoch 20: Misclassified = 3

Epoch 21: Misclassified = 3

Epoch 22: Misclassified = 3

Epoch 23: Misclassified = 3

Epoch 24: Misclassified = 3

Epoch 25: Misclassified = 3

Epoch 26: Misclassified = 3

Epoch 27: Misclassified = 3

Epoch 28: Misclassified = 3

Epoch 29: Misclassified = 3

Epoch 30: Misclassified = 3

Epoch 31: Misclassified = 3

Epoch 32: Misclassified = 3

Epoch 33: Misclassified = 3

Epoch 34: Misclassified = 3

Epoch 35: Misclassified = 3

Epoch 36: Misclassified = 3

Epoch 37: Misclassified = 3

Epoch 38: Misclassified = 3

Epoch 39: Misclassified = 3

Epoch 40: Misclassified = 3

Epoch 41: Misclassified = 2

Epoch 42: Misclassified = 2

Epoch 43: Misclassified = 2

Epoch 44: Misclassified = 1

Epoch 45: Misclassified = 1

Epoch 46: Misclassified = 1

Epoch 47: Misclassified = 1

Epoch 48: Misclassified = 0

Epoch 49: Misclassified = 0

Epoch 50: Misclassified = 0

Epoch 51: Misclassified = 0

Epoch 52: Misclassified = 0

Epoch 53: Misclassified = 0

Epoch 54: Misclassified = 0

Epoch 55: Misclassified = 0

Epoch 56: Misclassified = 0

Epoch 57: Misclassified = 0

Epoch 58: Misclassified = 0

Epoch 59: Misclassified = 0

Epoch 60: Misclassified = 0

Epoch 61: Misclassified = 0

Epoch 62: Misclassified = 0

Epoch 63: Misclassified = 0

Epoch 64: Misclassified = 0

Epoch 65: Misclassified = 0

Epoch 66: Misclassified = 0

Epoch 67: Misclassified = 0

Epoch 68: Misclassified = 0

Epoch 69: Misclassified = 0

Epoch 70: Misclassified = 0

Epoch 71: Misclassified = 0

Epoch 72: Misclassified = 0

Epoch 73: Misclassified = 0

Epoch 74: Misclassified = 0

Epoch 75: Misclassified = 0

Epoch 76: Misclassified = 0

Epoch 77: Misclassified = 0

Epoch 78: Misclassified = 0

Epoch 79: Misclassified = 0

Epoch 80: Misclassified = 0

Test Misclassified = 2

Accuracy for train:test ratio 0.7:0.3 is 0.666667

Prediction for e19: skips

Prediction for e20: skips

Epoch 1: Misclassified = 3

Epoch 2: Misclassified = 3

Epoch 3: Misclassified = 3

Epoch 4: Misclassified = 3

Epoch 5: Misclassified = 3

Epoch 6: Misclassified = 3

Epoch 7: Misclassified = 3

Epoch 8: Misclassified = 3

Epoch 9: Misclassified = 3

Epoch 10: Misclassified = 3

Epoch 11: Misclassified = 3

Epoch 12: Misclassified = 3

Epoch 13: Misclassified = 3

Epoch 14: Misclassified = 3

Epoch 15: Misclassified = 3

Epoch 16: Misclassified = 3

Epoch 17: Misclassified = 3

Epoch 18: Misclassified = 3

Epoch 19: Misclassified = 3

Epoch 20: Misclassified = 3

Epoch 21: Misclassified = 3

Epoch 22: Misclassified = 3

Epoch 23: Misclassified = 3

Epoch 24: Misclassified = 3

Epoch 25: Misclassified = 3

Epoch 26: Misclassified = 3

Epoch 27: Misclassified = 3

Epoch 28: Misclassified = 3

Epoch 29: Misclassified = 3

Epoch 30: Misclassified = 3

Epoch 31: Misclassified = 3

Epoch 32: Misclassified = 3

Epoch 33: Misclassified = 3

Epoch 34: Misclassified = 3

Epoch 35: Misclassified = 3

Epoch 36: Misclassified = 3

Epoch 37: Misclassified = 2

Epoch 38: Misclassified = 1

Epoch 39: Misclassified = 1

Epoch 40: Misclassified = 1

Epoch 41: Misclassified = 0

Epoch 42: Misclassified = 0

Epoch 43: Misclassified = 0

Epoch 44: Misclassified = 0

Epoch 45: Misclassified = 0

Epoch 46: Misclassified = 0

Epoch 47: Misclassified = 0

Epoch 48: Misclassified = 0

Epoch 49: Misclassified = 0

Epoch 50: Misclassified = 0

Epoch 51: Misclassified = 0

Epoch 52: Misclassified = 0

Epoch 53: Misclassified = 0

Epoch 54: Misclassified = 0

Epoch 55: Misclassified = 0

Epoch 56: Misclassified = 0

Epoch 57: Misclassified = 0

Epoch 58: Misclassified = 0

Epoch 59: Misclassified = 0

Epoch 60: Misclassified = 0

Epoch 61: Misclassified = 0

Epoch 62: Misclassified = 0

Epoch 63: Misclassified = 0

Epoch 64: Misclassified = 0

Epoch 65: Misclassified = 0

Epoch 66: Misclassified = 0

Epoch 67: Misclassified = 0

Epoch 68: Misclassified = 0

Epoch 69: Misclassified = 0

Epoch 70: Misclassified = 0

Epoch 71: Misclassified = 0

Epoch 72: Misclassified = 0

Epoch 73: Misclassified = 0

Epoch 74: Misclassified = 0

Epoch 75: Misclassified = 0

Epoch 76: Misclassified = 0

Epoch 77: Misclassified = 0

Epoch 78: Misclassified = 0

Epoch 79: Misclassified = 0

Epoch 80: Misclassified = 0

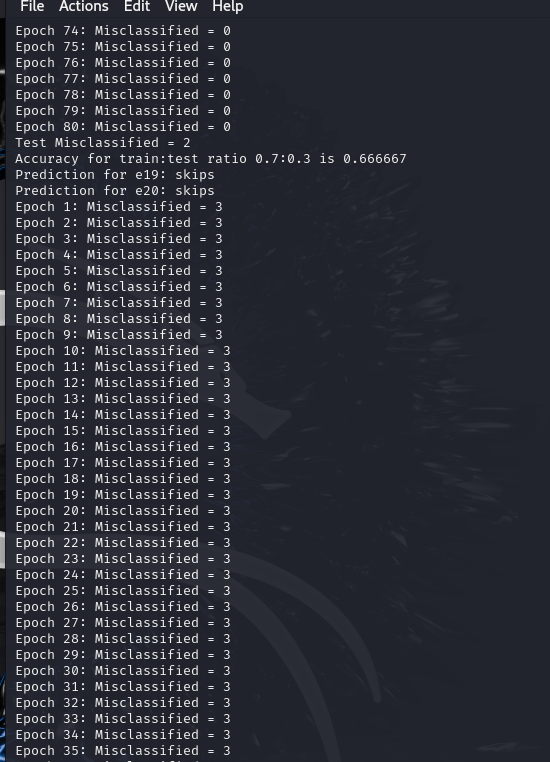
Test Misclassified = 2

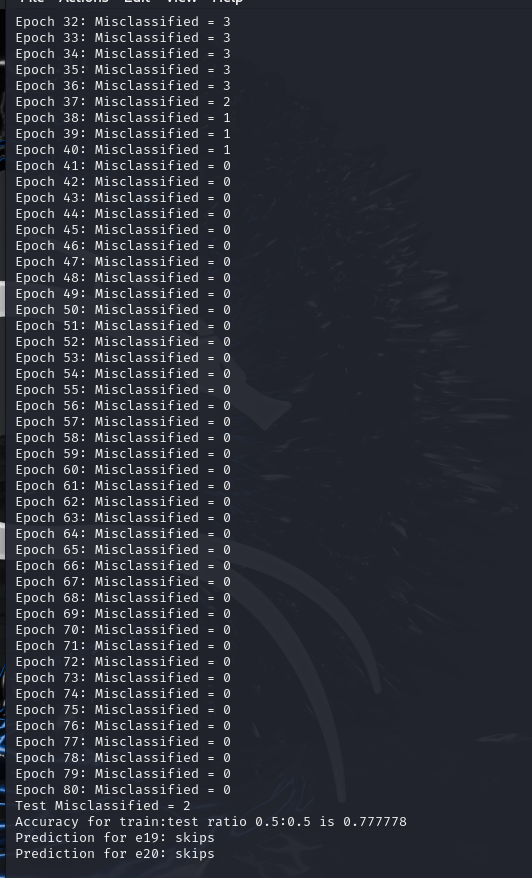
Accuracy for train:test ratio 0.5:0.5 is 0.777778

Prediction for e19: skips

Prediction for e20: skips

Screenshots:





a. Misclassification Analysis

Initial Misclassification:

From the output, at Epoch 1, the network has 5 misclassified examples. This represents the initial misclassification count.

Misclassification After 40 Iterations:

At Epoch 40, the output shows that the misclassified count is 1. This indicates that the network has reduced its error significantly by the 40th epoch.

Misclassification After 80 Iterations:

At Epoch 80, the misclassification count is 0. This means that by the end of 80 epochs, the model has correctly classified all examples in the training set, achieving perfect accuracy on the training data.

b. Effect of Different Step Sizes - Learning Rates

You are asked to test different values for the learning rate n (step sizes) and observe the effect on convergence. Let’s discuss the general observations based on different learning rates:

1. Learning Rate 𝜂 = 0.1 :

With a smaller learning rate, the network makes smaller adjustments to weights in each iteration. Convergence will be slow, so it may require more epochs to reduce misclassifications. The network will likely learn slowly and might not reach 0 misclassifications within 80 epochs.

1. Learning Rate 𝜂 = 1.0:

This is typically a moderate learning rate and might strike a balance between stability and speed of convergence. The network is likely to converge faster compared to 𝜂 = 0.1 and it may achieve 0 misclassifications within 80 epochs. As we observed in the output you provided. This suggests the learning rate of 0.7 or 1.0 is appropriate for this problem.

1. Learning Rate 𝜂 = 5.0:

A high learning rate can cause the network to make large adjustments to the weights. The network might rotate around the solution which results in failing to converge or even diverging resulting in increase in misclassifications instead of them decreasing. If you test this rate, you may observe unstable learning behavior and inconsistent misclassification counts.