Computer Intelligence Lab - CS354N

Weekly Assignment Report

Question 1:

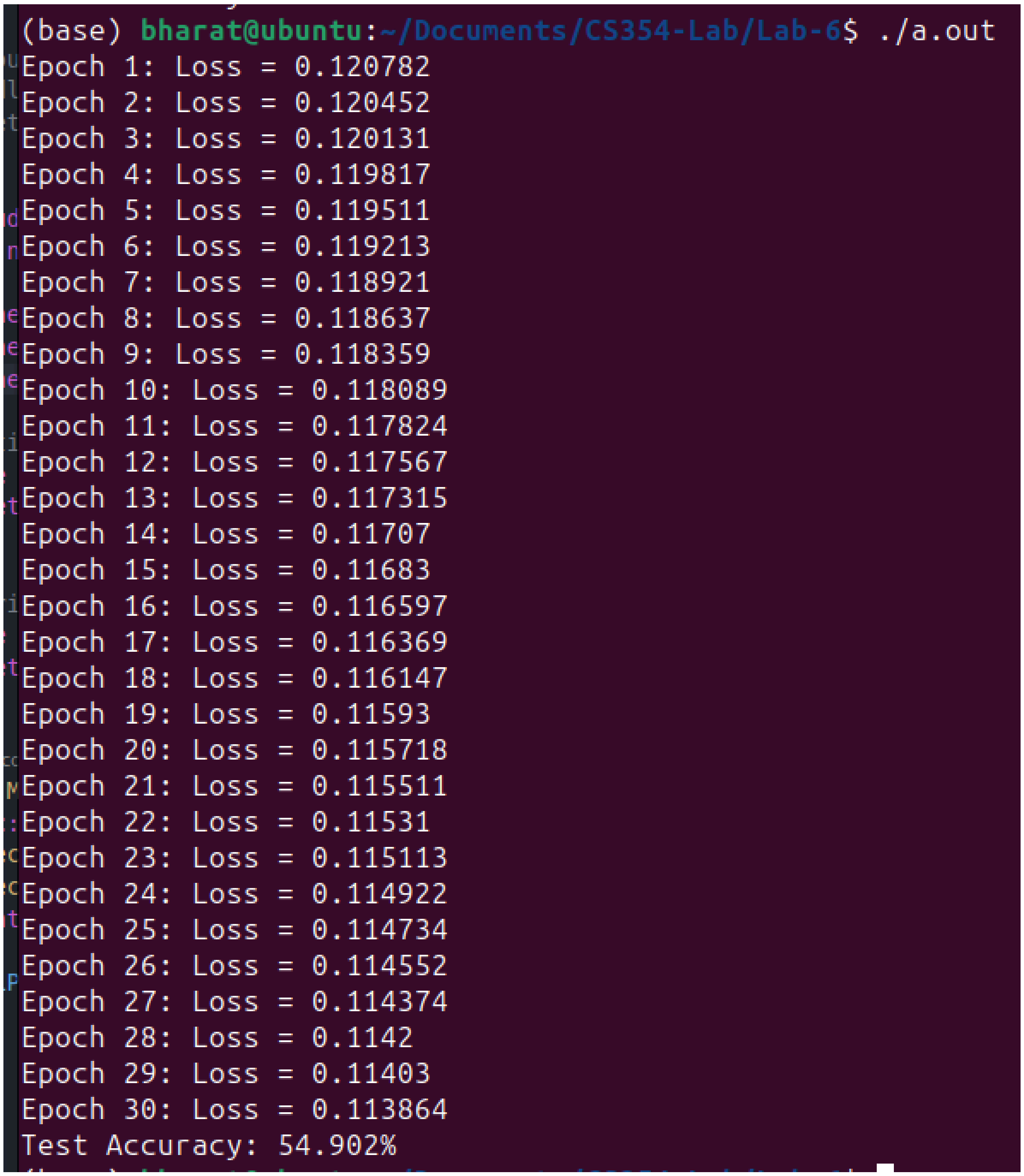
```
C/C++

#include <bits/stdc++.h>

using namespace std;


#define NUM_EPOCHS 30

#define LEARNING_RATE 0.0001

#define NUM_HIDDEN_NEURONS 10


// Activation function

inline double activate(double value) {

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

}


// Derivative of activation function

inline double activation_derivative(double value) {

    return value * (1 - value);

}


class MLPClassifier {

public:
```

```
vector<vector<double>> input_hidden_weights;

vector<double> hidden_output_weights;

int num_inputs, num_hidden, num_outputs;


MLPClassifier(int num_inputs, int num_hidden, int num_outputs) {

    this->num_inputs = num_inputs;

    this->num_hidden = num_hidden;

    this->num_outputs = num_outputs;


    input_hidden_weights.resize(num_inputs, vector<double>(num_hidden));

    hidden_output_weights.resize(num_hidden);


    random_device rd;

    mt19937 generator(rd());

    uniform_real_distribution<double> weight_dist(-1, 1);


    for (auto &layer : input_hidden_weights)

        for (auto &weight : layer)

            weight = weight_dist(generator);


    for (auto &weight : hidden_output_weights)

        weight = weight_dist(generator);

}
```

```
void train(vector<vector<double>> &train_data, vector<int> &train_labels) {

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

        double total_loss = 0.0;

        for (size_t sample = 0; sample < train_data.size(); ++sample) {

            vector<double> hidden_layer(num_hidden, 0.0);

            double output_layer = 0.0;



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

                for (int k = 0; k < num_inputs; ++k)

                    hidden_layer[j] += train_data[sample][k] *
input_hidden_weights[k][j];

                hidden_layer[j] = activate(hidden_layer[j]);

            }



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

                output_layer += hidden_layer[j] * hidden_output_weights[j];

            output_layer = activate(output_layer);



            double error = train_labels[sample] - output_layer;

            total_loss += 0.5 * error * error;
```

```
                double output_delta = error *
activation_derivative(output_layer);

                vector<double> hidden_deltas(num_hidden);


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

                    hidden_deltas[j] = output_delta * hidden_output_weights[j] *
activation_derivative(hidden_layer[j]);


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

                    hidden_output_weights[j] += LEARNING_RATE * output_delta *
hidden_layer[j];


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

                    for (int k = 0; k < num_inputs; ++k)

                        input_hidden_weights[k][j] += LEARNING_RATE *
hidden_deltas[j] * train_data[sample][k];

            }

            cout << "Epoch " << epoch + 1 << ": Loss = " << total_loss /
train_data.size() << endl;

        }

    }


    int classify(vector<double> &sample) {

        vector<double> hidden_layer(num_hidden, 0.0);
        double output_layer = 0.0;

        for (int j = 0; j < num_hidden; ++j) {
            for (int k = 0; k < num_inputs; ++k)
                hidden_layer[j] += sample[k] * input_hidden_weights[k][j];
            hidden_layer[j] = activate(hidden_layer[j]);
        }

        for (int j = 0; j < num_hidden; ++j)
            output_layer += hidden_layer[j] * hidden_output_weights[j];
        output_layer = activate(output_layer);

        return output_layer > 0.5 ? 1 : 0;
    }
};

vector<vector<double>> read_dataset(const string &file_path, vector<int>
&labels) {
    ifstream file(file_path);
    vector<vector<double>> dataset;
    string line;

    while (getline(file, line)) {
        stringstream ss(line);
        vector<double> features(8);
        for (int i = 0; i < 8; ++i)
            ss >> features[i], ss.ignore();

        int label;
        ss >> label;
        labels.push_back(label);
        dataset.push_back(features);
    }

    for (int i = 0; i < 8; ++i) {
        double max_value = 0.0;
        for (auto &sample : dataset)
            max_value = max(max_value, sample[i]);
        for (auto &sample : dataset)
            sample[i] /= max_value;
    }
    return dataset;
}

int main() {
    vector<int> train_labels, test_labels;
    vector<vector<double>> train_data = read_dataset("diabetes_train.csv",
train_labels);

    vector<vector<double>> test_data = read_dataset("diabetes_test.csv",
test_labels);


    MLPClassifier model(8, NUM_HIDDEN_NEURONS, 1);

    model.train(train_data, train_labels);


    int correct_predictions = 0;

    for (size_t i = 0; i < test_data.size(); ++i)

        if (model.classify(test_data[i]) == test_labels[i])

            ++correct_predictions;


    cout << "Test Accuracy: " << (double)correct_predictions / test_data.size()
* 100 << "%" << endl;

    return 0;

}
```

Output:



```
(base) bharat@ubuntu:~/Documents/CS354-Lab/Lab-6$ ./a.out
Epoch 1: Loss = 0.120782
Epoch 2: Loss = 0.120452
Epoch 3: Loss = 0.120131
Epoch 4: Loss = 0.119817
Epoch 5: Loss = 0.119511
Epoch 6: Loss = 0.119213
Epoch 7: Loss = 0.118921
Epoch 8: Loss = 0.118637
Epoch 9: Loss = 0.118359
Epoch 10: Loss = 0.118089
Epoch 11: Loss = 0.117824
Epoch 12: Loss = 0.117567
Epoch 13: Loss = 0.117315
Epoch 14: Loss = 0.11707
Epoch 15: Loss = 0.11683
Epoch 16: Loss = 0.116597
Epoch 17: Loss = 0.116369
Epoch 18: Loss = 0.116147
Epoch 19: Loss = 0.11593
Epoch 20: Loss = 0.115718
Epoch 21: Loss = 0.115511
Epoch 22: Loss = 0.11531
Epoch 23: Loss = 0.115113
Epoch 24: Loss = 0.114922
Epoch 25: Loss = 0.114734
Epoch 26: Loss = 0.114552
Epoch 27: Loss = 0.114374
Epoch 28: Loss = 0.1142
Epoch 29: Loss = 0.11403
Epoch 30: Loss = 0.113864
Test Accuracy: 54.902%
```

Question 2:

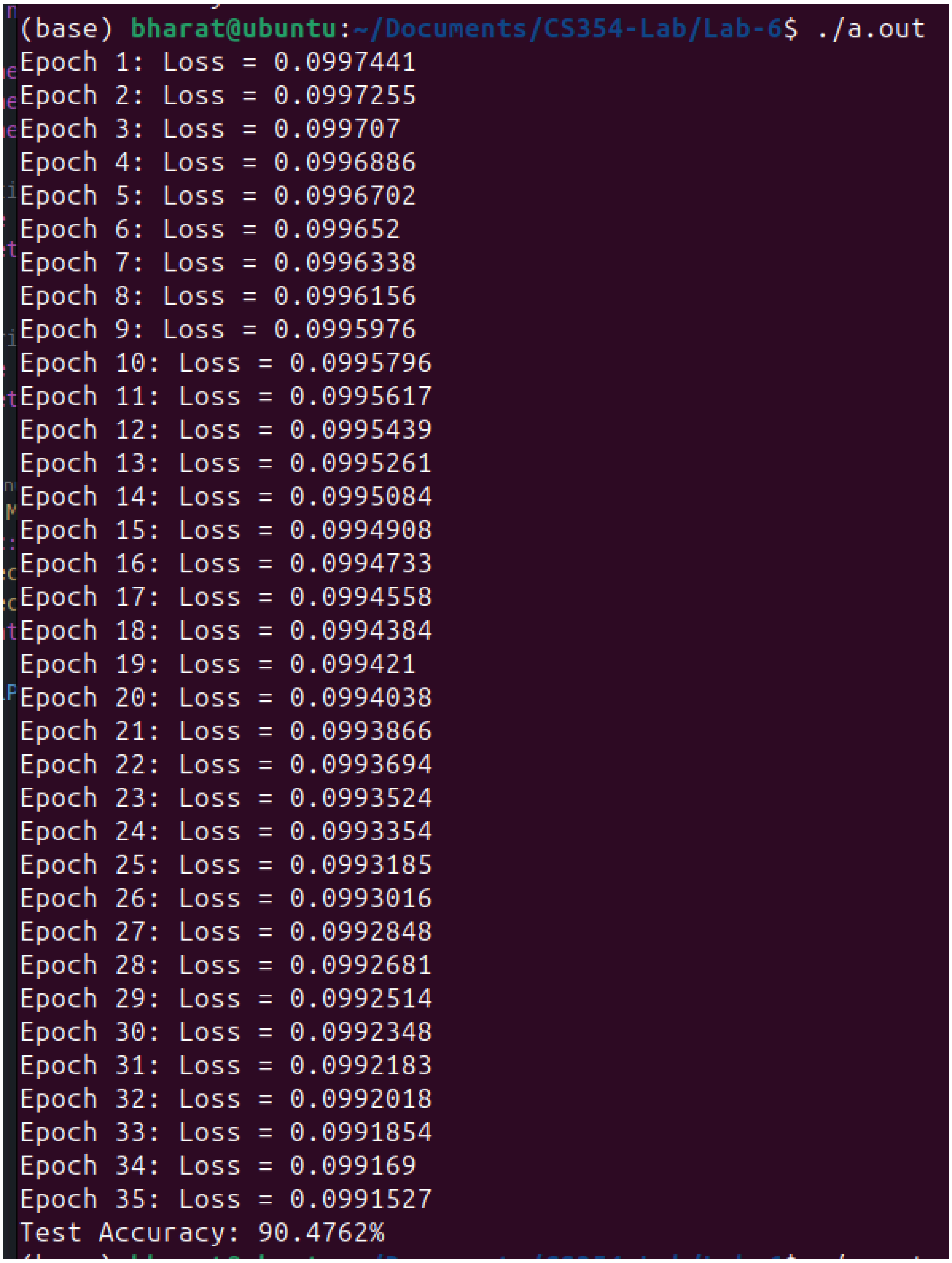
```
C/C++
#include <bits/stdc++.h>
using namespace std;


#define NUM_EPOCHS 35
#define LEARNING_RATE 0.0001
#define NUM_HIDDEN_NEURONS 15


// Activation function
inline double activate(double value) {
    return 1.0 / (1.0 + exp(-value));
}


// Derivative of activation function
inline double activation_derivative(double value) {
    return value * (1 - value);
}


class MLPClassifier {
public:
    vector<vector<double>> input_hidden_weights;
    vector<double> hidden_output_weights;
```

```cpp
int num_inputs, num_hidden, num_outputs;

MLPClassifier(int num_inputs, int num_hidden, int num_outputs) {
    this->num_inputs = num_inputs;
    this->num_hidden = num_hidden;
    this->num_outputs = num_outputs;

    input_hidden_weights.resize(num_inputs, vector<double>(num_hidden));
    hidden_output_weights.resize(num_hidden);

    random_device rd;
    mt19937 generator(rd());
    uniform_real_distribution<double> weight_dist(-1, 1);

    for (auto &layer : input_hidden_weights)
        for (auto &weight : layer)
            weight = weight_dist(generator);

    for (auto &weight : hidden_output_weights)
        weight = weight_dist(generator);
}

void train(vector<vector<double>> &train_data, vector<int> &train_labels) {
        for (int epoch = 0; epoch < NUM_EPOCHS; ++epoch) {

            double total_loss = 0.0;

            for (size_t sample = 0; sample < train_data.size(); ++sample) {

                vector<double> hidden_layer(num_hidden, 0.0);

                double output_layer = 0.0;


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

                    for (int k = 0; k < num_inputs; ++k)

                        hidden_layer[j] += train_data[sample][k] *
input_hidden_weights[k][j];

                    hidden_layer[j] = activate(hidden_layer[j]);

                }


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

                    output_layer += hidden_layer[j] * hidden_output_weights[j];

                output_layer = activate(output_layer);


                double error = train_labels[sample] - output_layer;

                total_loss += 0.5 * error * error;


                double output_delta = error *
activation_derivative(output_layer);

                vector<double> hidden_deltas(num_hidden);
                for (int j = 0; j < num_hidden; ++j)
                    hidden_deltas[j] = output_delta * hidden_output_weights[j] *
activation_derivative(hidden_layer[j]);

                for (int j = 0; j < num_hidden; ++j)
                    hidden_output_weights[j] += LEARNING_RATE * output_delta *
hidden_layer[j];

                for (int j = 0; j < num_hidden; ++j)
                    for (int k = 0; k < num_inputs; ++k)
                        input_hidden_weights[k][j] += LEARNING_RATE *
hidden_deltas[j] * train_data[sample][k];
            }
            cout << "Epoch " << epoch + 1 << ": Loss = " << total_loss /
train_data.size() << endl;
        }
    }

    int classify(vector<double> &sample) {
        vector<double> hidden_layer(num_hidden, 0.0);
        double output_layer = 0.0;

        for (int j = 0; j < num_hidden; ++j) {
            for (int k = 0; k < num_inputs; ++k)
                hidden_layer[j] += sample[k] * input_hidden_weights[k][j];
            hidden_layer[j] = activate(hidden_layer[j]);
        }

        for (int j = 0; j < num_hidden; ++j)
            output_layer += hidden_layer[j] * hidden_output_weights[j];
        output_layer = activate(output_layer);

        return output_layer > 0.5 ? 1 : 0;
    }
};

vector<vector<double>> read_dataset(const string &file_path, vector<int>
&labels) {
    ifstream file(file_path);
    vector<vector<double>> dataset;
    string line;

    while (getline(file, line)) {
        stringstream ss(line);
        vector<double> features(4);
        for (int i = 0; i < 4; ++i)
```

```
            ss >> features[i], ss.ignore();


        string label;

        ss >> label;

        labels.push_back(label == "setosa" ? 0 : 1);

        dataset.push_back(features);

    }


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

        double max_value = 0.0;

        for (auto &sample : dataset)

            max_value = max(max_value, sample[i]);

        for (auto &sample : dataset)

            sample[i] /= max_value;

    }

    return dataset;

}


int main() {

    vector<int> train_labels, test_labels;

    vector<vector<double>> train_data = read_dataset("iris_train.csv",
train_labels);
```

```
    vector<vector<double>> test_data = read_dataset("iris_test.csv",
test_labels);


    MLPClassifier model(4, NUM_HIDDEN_NEURONS, 1);

    model.train(train_data, train_labels);


    int correct_predictions = 0;

    for (size_t i = 0; i < test_data.size(); ++i)

        if (model.classify(test_data[i]) == test_labels[i])

            ++correct_predictions;


    cout << "Test Accuracy: " << (double)correct_predictions / test_data.size()
* 100 << "%" << endl;

    return 0;

}
```

Output:



```
(base) bharat@ubuntu:~/Documents/CS354-Lab/Lab-6$ ./a.out
Epoch 1: Loss = 0.0997441
Epoch 2: Loss = 0.0997255
Epoch 3: Loss = 0.099707
Epoch 4: Loss = 0.0996886
Epoch 5: Loss = 0.0996702
Epoch 6: Loss = 0.099652
Epoch 7: Loss = 0.0996338
Epoch 8: Loss = 0.0996156
Epoch 9: Loss = 0.0995976
Epoch 10: Loss = 0.0995796
Epoch 11: Loss = 0.0995617
Epoch 12: Loss = 0.0995439
Epoch 13: Loss = 0.0995261
Epoch 14: Loss = 0.0995084
Epoch 15: Loss = 0.0994908
Epoch 16: Loss = 0.0994733
Epoch 17: Loss = 0.0994558
Epoch 18: Loss = 0.0994384
Epoch 19: Loss = 0.099421
Epoch 20: Loss = 0.0994038
Epoch 21: Loss = 0.0993866
Epoch 22: Loss = 0.0993694
Epoch 23: Loss = 0.0993524
Epoch 24: Loss = 0.0993354
Epoch 25: Loss = 0.0993185
Epoch 26: Loss = 0.0993016
Epoch 27: Loss = 0.0992848
Epoch 28: Loss = 0.0992681
Epoch 29: Loss = 0.0992514
Epoch 30: Loss = 0.0992348
Epoch 31: Loss = 0.0992183
Epoch 32: Loss = 0.0992018
Epoch 33: Loss = 0.0991854
Epoch 34: Loss = 0.099169
Epoch 35: Loss = 0.0991527
Test Accuracy: 90.4762%
```

Observation:

The model's accuracy suggests it might not be learning well or the dataset is complex. A few possible reasons:

- **Too few hidden nodes:** Since the data is more complex and there are only 5-15 hidden neurons, the model is not able to learn complex patterns, hence increasing them will help in increasing the accuracy of the model.
- **Suboptimal learning rate:** Learning rate of 0.0001 is pretty low, increasing it might help in increasing the accuracy.
- **More epochs:** Number of epochs is pretty low considering data size, increasing it will help in learning the complex patterns for the model and thus, getting higher accuracy.
- **Random Weights Initialization:** Sometimes, randomly initialising the weights might help in getting high accuracy, as can be seen from the output of the question, where we got an accuracy of 90.47%, a single time while the weights were optimally initialized, while all the other time the accuracy was 54-55%.