**PRACTICAL 5**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name:** | Harsh Shah | **Semester:** | VI | **Division:** | 6 |
| **Roll No.:** | 21BCP359 | **Date:** |  | **Batch:** | G11 |
| **Aim:** | Given a C++ code, identify the algorithm implemented through the code. Also document the code. | | | | |

**Program**

#include <iostream>

#include <cstdio>

#include <cstdlib>

#include <cstring>

#include <ctime>

#include <cmath>

using namespace std;

// GLOBAL FILE NAME

char file\_name[9], file\_name\_inf[14], file\_name\_wgt[14], file\_name\_rst[14];

char file\_name\_out[14], file\_name\_dat[14];

// Class representing a matrix

class matrix {

int row, col;

public:

float mat[15][15];

matrix() {

row = 0;

col = 0;

}

void set(int, int);

int getrows() {

return row;

}

int getcols() {

return col;

}

void getdata();

FILE \*fgetdata(FILE \*);

void displaydata();

void displaydat();

FILE \*fputdata(FILE \*);

FILE \*fputdat(FILE \*);

matrix operator+(matrix);

matrix operator-();

matrix operator\*(matrix);

matrix operator\*(float);

};

// Set the dimensions of the matrix

void matrix::set(int i, int j) {

row = i;

col = j;

}

// Read matrix data from file

FILE \*matrix::fgetdata(FILE \*fmat) {

char line;

int i, j;

fscanf(fmat, "%d%d", &(row), &(col));

for (i = 1; i <= row; i++)

for (j = 1; j <= col; j++)

fscanf(fmat, "%f", &(mat[i][j]));

return (fmat);

}

// Read matrix data from user input

void matrix::getdata() {

int i, j;

cout << "Enter the size of the matrix:";

cin >> row >> col;

for (i = 1; i <= row; i++)

for (j = 1; j <= col; j++) {

cout << "element [" << i << "] [ " << j << " ] ";

cin >> mat[i][j];

}

}

// Display matrix data

void matrix::displaydata() {

int i, j;

for (i = 1; i <= row; i++, printf("\n\r"))

for (j = 1; j <= col; j++, printf("\t"))

printf("\t\t%10.2f", mat[i][j]);

}

// Display matrix dimensions

void matrix::displaydat() {

int i;

cout << row;

}

// Write matrix data to file

FILE \*matrix::fputdata(FILE \*fmat) {

int i, j;

fprintf(fmat, "%d\n%d\n", row, col);

for (i = 1; i <= row; i++)

for (j = 1; j <= col; j++)

fprintf(fmat, "%f\n", mat[i][j]);

return (fmat);

}

// Write matrix dimensions to file

FILE \*matrix::fputdat(FILE \*fmat) {

int i;

fprintf(fmat, "%d", row);

return (fmat);

}

// Overloaded operator for matrix addition

matrix matrix::operator+(matrix m) {

matrix temp;

int i, j;

if ((row == m.row) && (col == m.col))

for (i = 1; i <= row; i++)

for (j = 1; j <= col; j++)

temp.mat[i][j] = mat[i][j] + m.mat[i][j];

else {

cout << "The addition of the matrices is not possible";

exit(1);

}

temp.row = row;

temp.col = col;

return (temp);

}

// Overloaded operator for matrix transposition

matrix matrix::operator-() {

matrix temp;

int i, j;

temp.row = col;

temp.col = row;

for (i = 1; i <= col; i++)

for (j = 1; j <= row; j++)

temp.mat[i][j] = mat[j][i];

return (temp);

}

// Overloaded operator for matrix multiplication

matrix matrix::operator\*(matrix m) {

matrix temp;

int i, j, k;

if (col == m.row) {

for (i = 1; i <= row; i++)

for (j = 1; j <= m.col; j++) {

temp.mat[i][j] = 0;

for (k = 1; k <= col; k++)

temp.mat[i][j] = temp.mat[i][j] + (mat[i][k] \* m.mat[k][j]);

}

} else {

cout << "The multiplication of the matrices is not possible";

exit(1);

}

temp.row = row;

temp.col = m.col;

return (temp);

}

// Overloaded operator for scalar multiplication with matrix

matrix matrix::operator\*(float svalue) {

matrix temp;

int i, j;

for (i = 1; i <= row; i++)

for (j = 1; j <= col; j++)

temp.mat[i][j] = mat[i][j] \* svalue;

temp.row = row;

temp.col = col;

return (temp);

}

// Class representing the training process of the neural network

class training {

FILE \*fin, \*fout, \*fwt;

matrix Input[5], Output[5], Weights[5], dWeights[5], d, e, T;

float alpha, eta, err, theta, lamda, error;

int TotalLayers, HiddenLayers, l[5], ntest, iterates;

long filepos;

public:

training();

void readinputs();

void printing();

void initweights();

void initdweights();

void train();

void io\_values();

void backpropagate();

void errors();

void chgweights();

void newweights();

~training();

};

// Constructor for training class

training::training() {

// Open files for input and output

if ((fin = fopen(file\_name\_dat, "r")) == NULL) exit(1);

if ((fout = fopen(file\_name\_out, "w")) == NULL) exit(1);

if ((fwt = fopen(file\_name\_wgt, "w")) == NULL) exit(1);

readinputs();

printing();

initweights();

initdweights();

train();

}

// Function to read input parameters for training

void training::readinputs() {

int i;

error = 0;

char line;

fscanf(fin, "%d", &HiddenLayers); // Get number of hidden layers

TotalLayers = HiddenLayers + 1; // Calculate total number of layers

for (i = 0; i <= TotalLayers; i++)

fscanf(fin, "%d", &l[i]);

fscanf(fin, "%f%f%f%f%f", &alpha, &err, &eta, &theta, &lamda);

fscanf(fin, "%d%d", &ntest, &iterates);

filepos = ftell(fin);

}

// Function to print input parameters for training

void training::printing() {

// Print parameters to output file

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

fprintf(fout, "\nNumber of Neurons in layer[%d]=%d", i + 1, l[i]);

fprintf(fout, "\nAlpha value(Momentum factor): %f", alpha);

fprintf(fout, "\nError constant : %f", err);

fprintf(fout, "\nLearning rate : %f", eta);

fprintf(fout, "\nThreshold value : %f", theta);

fprintf(fout, "\nScaling Parameter: %f", lamda);

fprintf(fout, "\nNo of Training data : %d", ntest);

fprintf(fout, "\nMaximum Iteration : %d", iterates);

system("cls");

// Print parameters to console

printf("\n\n\n");

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

printf("\n\t\tNumber of Neurons in layer[%d]=%d", i + 1, l[i]);

printf("\n\n\t\tAlpha value(Momentum factor): %f", alpha);

printf("\n\t\tError constant : %f", err);

printf("\n\t\tLearning rate : %f", eta);

printf("\n\t\tThreshold value : %f", theta);

printf("\n\t\tScaling Parameter : %f", lamda);

printf("\n\t\tNo of Training data : %d", ntest);

printf("\n\t\tMaximum Iteration : %d", iterates);

cin.get();

}

// Function to initialize weights randomly

void training::initweights() {

srand(2000);

srand(time(0));

for (int k = 0; k < TotalLayers; k++) {

Weights[k].set(l[k], l[k + 1]);

for (int i = 1; i <= l[k]; i++)

for (int j = 1; j <= l[k + 1]; j++)

Weights[k].mat[i][j] = ((float)rand() / 32767) - 0.5;

fprintf(fout, "\nWeights[%d]:", k);

Weights[k].fputdata(fout);

}

}

// Function to initialize difference in weights

void training::initdweights() {

for (int k = 0; k < TotalLayers; k++) {

dWeights[k].set(l[k], l[k + 1]);

for (int i = 1; i <= l[k]; i++)

for (int j = 1; j <= l[k + 1]; j++)

dWeights[k].mat[i][j] = 0.0;

}

}

// Function to perform neural network training

void training::train() {

int k;

for (int jtr = 1; jtr <= iterates; jtr++) {

error = 0.0;

fseek(fin, filepos, SEEK\_SET);

cout << "\nIteration Number: " << jtr << endl;

for (int itr = 1; itr <= ntest; itr++) {

Input[0].fgetdata(fin);

T.fgetdata(fin);

cout << "\rTraining Data Number: " << itr;

io\_values();

backpropagate();

errors();

chgweights();

newweights();

}

fprintf(fout, " %10.3E\n", error / ntest);

}

cin.get();

for (k = 0; k < TotalLayers; k++)

fwt = Weights[k].fputdata(fwt);

}

// Function to calculate input/output values of neurons

void training::io\_values() {

Output[0] = Input[0];

for (int m = 0; m <= TotalLayers - 1; m++) {

Input[m + 1] = -Weights[m] \* Output[m];

Output[m + 1].set(l[m + 1], 1);

for (int i = 1; i <= l[m + 1]; i++)

Output[m + 1].mat[i][1] = 1.0 / (1.0 + exp(-lamda \* (Input[m + 1].mat[i][1] + theta)));

}

}

// Function to perform backpropagation

void training::backpropagate() {

d.set(l[TotalLayers], 1);

for (int i = 1; i <= l[TotalLayers]; i++)

d.mat[i][1] = Output[TotalLayers].mat[i][1] \* (1 - Output[TotalLayers].mat[i][1]) \* (T.mat[i][1] - Output[TotalLayers].mat[i][1]);

dWeights[TotalLayers - 1] = (dWeights[TotalLayers - 1] \* alpha) + ((Output[TotalLayers - 1] \* -d) \* eta);

}

// Function to calculate errors

void training::errors() {

float sum = 0.0, x, y1, y2;

for (int j = 1; j <= l[TotalLayers]; j++) {

y1 = T.mat[j][1];

y2 = Output[TotalLayers].mat[j][1];

x = fabs(y1 - y2);

x = x \* x;

sum = sum + x;

}

sum = sqrt(sum / l[TotalLayers]);

error = error + sum;

cout << "\t\t Error =" << error;

}

// Function to calculate change in weights

void training::chgweights() {

int k;

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

k = TotalLayers - i - 1;

e = Weights[k] \* d;

d.set(l[k], 1);

for (int j = 1; j <= l[k]; j++) {

d.mat[j][1] = Output[k].mat[j][1] \* (1 - Output[k].mat[j][1]) \* e.mat[j][1];

}

dWeights[k - 1] = (dWeights[k - 1] \* alpha) + ((Output[k - 1] \* -d) \* eta);

}

}

// Function to update weights

void training::newweights() {

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

Weights[k] = Weights[k] + dWeights[k];

}

// Destructor for training class

training::~training() {

fclose(fin);

fclose(fout);

fclose(fwt);

}

// Class representing the inference process of the neural network

class inference {

FILE \*fin, \*fout, \*fwt;

matrix Input[5], Output[5], Weights[5], T, CalculatedErr, NoOfTest;

float alpha, eta, err, theta, x1, x2, lamda, Calerror;

int TotalLayers, ntest, l[10];

public:

inference();

void readinputs();

void initweights();

void i\_values();

void calculate();

void error();

~inference();

};

// Constructor for inference class

inference::inference() {

// Open files for input and output

if ((fin = fopen(file\_name\_inf, "r")) == NULL) exit(1);

if ((fout = fopen(file\_name\_rst, "w")) == NULL) exit(1);

if ((fwt = fopen(file\_name\_wgt, "r")) == NULL) exit(1);

readinputs();

initweights();

i\_values();

calculate();

error();

}

// Function to read input parameters for inference

void inference::readinputs() {

int i;

fscanf(fin, "%d", &TotalLayers); // Get number of hidden layers

for (i = 0; i <= TotalLayers; i++)

fscanf(fin, "%d", &l[i]); // Get number of neurons in each layer

fscanf(fin, "%f%f%f%f%f", &alpha, &err, &eta, &theta, &lamda); // Get other parameters

fscanf(fin, "%d", &ntest); // Get number of test cases

}

// Function to initialize weights for inference

void inference::initweights() {

for (int k = 0; k < TotalLayers; k++) {

Weights[k].fgetdata(fwt);

}

}

// Function to calculate input values for inference

void inference::i\_values() {

for (int itr = 1; itr <= ntest; itr++) {

Input[0].fgetdata(fin);

cout << "\rTesting Data Number: " << itr;

Output[0] = Input[0];

for (int m = 0; m <= TotalLayers - 1; m++) {

Input[m + 1] = -Weights[m] \* Output[m];

Output[m + 1].set(l[m + 1], 1);

for (int i = 1; i <= l[m + 1]; i++)

Output[m + 1].mat[i][1] = 1.0 / (1.0 + exp(-lamda \* (Input[m + 1].mat[i][1] + theta)));

}

Output[TotalLayers].displaydat();

}

}

// Function to perform calculation for inference

void inference::calculate() {

float sum = 0.0;

for (int i = 1; i <= ntest; i++) {

T.fgetdata(fin);

CalculatedErr = Output[TotalLayers] - T;

CalculatedErr = -CalculatedErr;

CalculatedErr = CalculatedErr \* CalculatedErr;

x1 = CalculatedErr.mat[1][1];

sum = sum + x1;

}

sum = sqrt(sum / ntest);

Calerror = sum;

}

// Function to calculate error for inference

void inference::error() {

printf("\nCalculated Error: %f", Calerror);

fprintf(fout, "%f", Calerror);

}

// Destructor for inference class

inference::~inference() {

fclose(fin);

fclose(fout);

}

// Main function

int main() {

strcpy(file\_name, "Nndat.dat");

strcpy(file\_name\_inf, "Nntst.dat");

strcpy(file\_name\_wgt, "Nnwgt.dat");

strcpy(file\_name\_out, "Nnout.dat");

strcpy(file\_name\_rst, "Nnres.dat");

training mlp1;

inference mlp2;

return 0;

}

**Explanation**

The algorithm implemented through the code is **Backpropagation for training a Multi-Layer Perceptron (MLP)** neural network.

Here's a breakdown of the code and its functionalities:

**Classes:**

* matrix: This class represents a matrix and provides methods for creating, manipulating, and displaying matrices.
* training: This class handles the training process of the MLP network. It includes methods for reading training data, initializing weights, performing backpropagation, calculating errors, and updating weights.
* inference: This class performs inference on the trained network. It reads test data, calculates the network's output, and compares it to the desired output.

**Training Process (training class):**

1. **Initialization:**

* Reads training data and network configuration from a file.
* Initializes weights and learning parameters.

1. **Iteration Loop:**

* Loops for a specified number of iterations.
* For each iteration:
* Loops for each training data point:
* Calculates the output of each layer using the forward pass.
* Performs backpropagation to calculate the error gradients.
* Updates the weights using the gradient descent algorithm with momentum.

1. **Weight Update:**

* Writes the final weights to a file.

**Inference Process (inference class):**

1. **Loading Configuration:**

* Reads network configuration and weights from files.

1. **Test Data Loop:**

* Loops for each test data point:
* Calculates the network's output using the forward pass.
* Compares the output to the desired output and calculates the error.
* Writes the calculated output, actual output, and error to a file.

**Overall, the code implements a backpropagation algorithm to train a multilayer perceptron neural network. The trained network can then be used for inference on new data.**