Alexander Ngo

CS 4613

Project 2: Handwritten Digit Recognition

**Filename and how to compile/run the program**

The source code is contained within 2 .cpp files and 1 .h file:

* main.cpp
* digit.cpp
* digit.h

The files must be linked on any IDE to be ran, or one can simply run the program via “Handwritten Digit Recognition.exe”. To change the number of neurons in the hidden layer, line 17 of main.cpp must be changed to accommodate for the new amount. After the program is executed, “Results.txt” should appear in the same directory.

After several minutes (time varies on size of hidden layer and amount of images used to train + test) the contents of “Results.txt” should be updated with the confusion matrix and several other significant pieces of information regarding the training and accuracy of the network. (The figures can be seen in “Confusion matrix and accuracy” below)

**Training Related Questions & Answers**

How were the weight values of the network initialized?

* The weights were randomized to be between -1.00 and 1.00 (with digits varying to the hundredths place). These values were used as they gave 201 different possible values that were optimal to use alongside the used sigmoid function.

How many iterations through the training did the algorithm perform?

* The algorithm iterated through around 2000 images on average before it was confident in classifying images correctly.

How did the network decide when to stop training?

* Ending the training session was based on the change in the mean squared error. After observing the trend of the mean squared error over a several runs of the program, it was found that the algorithm mostly converged when the mean squared error stopped decreasing in the hundredths place after several hundred iterations. i.e. When the mean squared error several hundred iterations ago – the current mean squared error < 0.01.

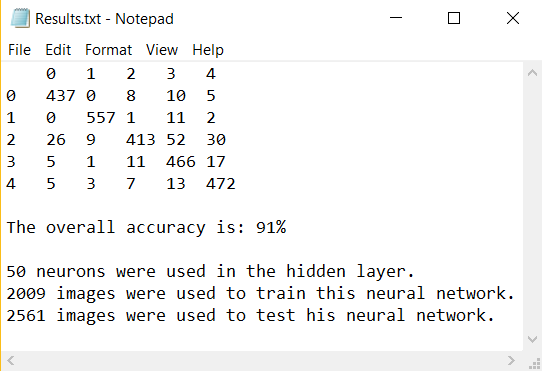
Based on the output values of the five output neurons, how did the network decide which digit to classify the input image into?

* The output values ranged between 0 and 1. Each neuron is associated with one of 5 possible values (0 to 4). The neuron with the maximum output, or the output closest to 1, was used to classify each image.

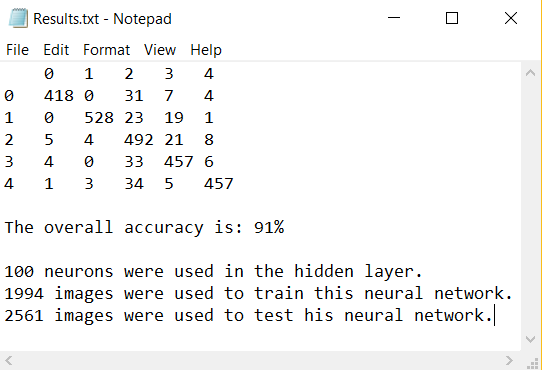
**Confusion matrix and accuracy**

Here are the results copied straight from the generated ‘Results.txt’ file for the network with the following amount of hidden layer neurons (rows = true classification, columns = output classification):

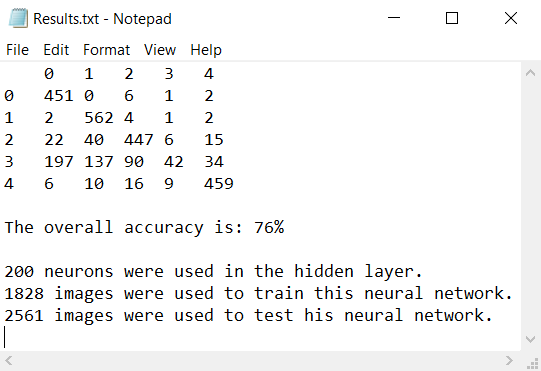
50



100



200



**Additional Comments**

The amount of time it takes to run the program and get results is largely based on the differing sizes of the hidden layer. It usually takes around 6 minutes to run with 50 hidden layer neurons, 10 minutes with 100 hidden layer neurons, and 30 minutes with 200 hidden layer neurons.

**Source Code**

\*The spacing might be different in this document

**Main.cpp**

// Alexander Ngo

// AI Project: Handwritten Digit Recognition

// Two-Layer Fully-Connected Perception Neural Network for recognizing digits from 0 to 4

#include "digit.h"

using namespace std;

using namespace digitrecognition;

int main() {

//dataset

int n\_train = 28038;

int n\_test = 2561;

//layers

int n\_inputs = 784; // 28x28 pixels

int n\_hidden = 50;

//caculation related variables

float alpha = 0.1f;

int random\_range = 1;

float bias = -1.0f;

Network\* network = new Network(n\_inputs, n\_hidden, alpha, random\_range, bias);

char trainFile[20] = "train\_images.raw";

char testFile[20] = "test\_images.raw";

FILE \*fp1;

ifstream trainLabel("train\_labels.txt");

FILE \*fp2;

ifstream testLabel("test\_labels.txt");

fp1 = fopen(trainFile, "rb");

fp2 = fopen(testFile, "rb");

//train the network

int i = 0;

while (i < n\_train && !network->done\_training) {

network->train(fp1, trainLabel);

i++;

}

cout << endl;

//test the network

i = 0;

while (i < n\_test) {

network->test(fp2, testLabel);

i++;

}

network->printResults(n\_test, n\_hidden);

fclose(fp1);

trainLabel.close();

fclose(fp2);

testLabel.close();

return 0;

}

**digit.h**

#ifndef DIGIT\_HPP

#define DIGIT\_HPP

#include <vector>

#include <iostream>

#include <fstream>

#include <math.h>

#include <random>

namespace digitrecognition {

class Neuron {

public:

Neuron(int inputs, int random\_range, float bias);

void activation();

std::vector<float> values;

std::vector<float> weights;

float bias;

float delta;

std::vector<float> epoch\_avg;

float output;

};

class Layer {

public:

std::vector<Neuron\*> neurons;

};

class Network {

public:

Network(int n\_inputs, int n\_hidden, float alpha, int random\_range, float bias);

void train(FILE\* image, std::ifstream& labels);

void read(FILE\* file);

void calculateCost(std::ifstream& labels);

void backpropagate();

void test(FILE\* image, std::ifstream& labels);

void classify(std::ifstream& labels);

void printResults(int test\_examples, int hidden);

int n\_inputs;

int n\_hidden;

int n\_output = 5;

float alpha;

int random\_range;

float cost;

float prev\_squared\_errors;

float mean\_squared\_error;

int trained\_examples = 0;

bool done\_training = false;

Layer hiddenLayer;

Layer outputLayer;

std::vector<float> inputVec;

std::vector<int> correctOutputs;

std::vector<float> recent\_mean\_sqaured\_errors;

std::vector<std::vector<int>> confusionMatrix;

};

float sigmoid(float x);

}

#endif

**digit.cpp**

#include "digit.h"

using namespace std;

namespace digitrecognition {

ofstream outfile("Results.txt");

float sigmoid(float x) {

// 1/ (1 + e^-x)

float result = 1 / (1 + (exp(-x)));

return result;

}

Network::Network(int n\_inputs, int n\_hidden, float alpha, int random\_range, float bias) : n\_inputs(n\_inputs), n\_hidden(n\_hidden), alpha(alpha), random\_range(random\_range) {

for (int i = 0; i < n\_hidden; i++) {

Neuron\* np = new Neuron(n\_inputs, random\_range, bias);

hiddenLayer.neurons.push\_back(np);

}

for (int i = 0; i < 5; i++) {

Neuron\* np = new Neuron(n\_hidden, random\_range, bias);

outputLayer.neurons.push\_back(np);

}

inputVec.insert(inputVec.end(), n\_inputs, 0.0f);

correctOutputs.insert(correctOutputs.end(), 5, 0);

vector<int> trueClass{ 0,0,0,0,0 };

confusionMatrix.push\_back(trueClass);

confusionMatrix.push\_back(trueClass);

confusionMatrix.push\_back(trueClass);

confusionMatrix.push\_back(trueClass);

confusionMatrix.push\_back(trueClass);

}

Neuron::Neuron(int inputs, int random\_range, float bias) : bias(bias) {

//Initializing random seed

std::random\_device rd;

std::mt19937 gen(rd());

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

//Generate a random weight between negative and positive random\_range (to the hundredths place)

std::uniform\_int\_distribution<> rnum(random\_range\*-100, random\_range \* 100);

float weight = (float)rnum(gen);

weight /= 100.0f;

weights.push\_back(weight);

values.push\_back(0.0f);

epoch\_avg.push\_back(0.0f);

}

}

void Neuron::activation() {

float weightedSum = 0;

//First compute weighted sum

for (size\_t i = 0; i < values.size(); i++) {

weightedSum += weights[i] \* values[i];

}

weightedSum += bias;

//The compute output using weighted sum and sigmoid function

output = sigmoid(weightedSum);

}

void Network::read(FILE\* file) {

short ch;

int i = 0;

while (i != n\_inputs) {

ch = getc(file);

inputVec[i] = ((float)((unsigned int)ch)) / 255.0f;

i++;

}

//Feed data from inputVector into each hiddenLayer neuron

for (int i = 0; i < n\_hidden; i++) {

for (int j = 0; j < n\_inputs; j++) {

hiddenLayer.neurons[i]->values[j] = inputVec[j];

}

}

}

void Network::calculateCost(ifstream& labels) {

labels >> correctOutputs[0] >> correctOutputs[1] >> correctOutputs[2] >> correctOutputs[3] >> correctOutputs[4];

cost = 0.0f;

for (int i = 0; i < 5; i++) {

cost += 0.5f \* pow((float)correctOutputs[i] - outputLayer.neurons[i]->output, 2);

}

prev\_squared\_errors += cost;

trained\_examples++;

mean\_squared\_error = prev\_squared\_errors/ trained\_examples;

//Add the mean sqaured error to the vector of recent mean squared errors and check if it changed much at all from the one 300 iterations ago

//If there's not much of a difference, we can stop training and go to testing

if (recent\_mean\_sqaured\_errors.size() < 300) {

recent\_mean\_sqaured\_errors.push\_back(mean\_squared\_error);

}

else {

//Check if we should stop training

//Not allowed to start checking until at least 1000 exampels have been used to train

if (recent\_mean\_sqaured\_errors[0] - mean\_squared\_error < 0.01f && trained\_examples > 1000) {

done\_training = true;

}

//Then move all iterations down the vector

recent\_mean\_sqaured\_errors.erase(recent\_mean\_sqaured\_errors.begin());

recent\_mean\_sqaured\_errors.push\_back(mean\_squared\_error);

}

}

void Network::backpropagate() {

//Start with output layer

for (int i = 0; i < 5; i++) {

//Calculate delta of output to hidden layer

float weightedSum = 0.0f;

for (int j = 0; j < n\_hidden; j++) {

weightedSum += outputLayer.neurons[i]->weights[j] \* outputLayer.neurons[i]->values[j];

}

float derived\_sigmoid = sigmoid(weightedSum) \* (1 - sigmoid(weightedSum));

outputLayer.neurons[i]->delta = ((float)correctOutputs[i] - outputLayer.neurons[i]->output) \* derived\_sigmoid;

for (int j = 0; j < n\_hidden; j++) {

outputLayer.neurons[i]->weights[j] += alpha \* outputLayer.neurons[i]->delta \* hiddenLayer.neurons[j]->output;

}

}

//Backpropagate to hidden layer

for (int i = 0; i < n\_hidden; i++) {

float weightedSum = 0.0f;

//Calculate derived\_sigmoid for delta value

for (int j = 0; j < n\_inputs; j++) {

weightedSum += hiddenLayer.neurons[i]->weights[j] \* hiddenLayer.neurons[i]->values[j];

}

float derived\_sigmoid = sigmoid(weightedSum) \* (1 - sigmoid(weightedSum));

float summation = 0.0f;

//Calculate summation of (weights to output layer \* corresponding output delta)

for (int k = 0; k < 5; k++) {

summation += outputLayer.neurons[k]->weights[i] \* outputLayer.neurons[k]->delta;

}

//Combined above two parts for delta

hiddenLayer.neurons[i]->delta = derived\_sigmoid \* summation;

for (int j = 0; j < n\_inputs; j++) {

hiddenLayer.neurons[i]->weights[j] += alpha \* hiddenLayer.neurons[i]->delta \* inputVec[j];

}

}

}

void Network::train(FILE\* image, std::ifstream& labels) {

read(image);

//Call activation function

for (int i = 0; i < n\_hidden; i++) {

hiddenLayer.neurons[i]->activation();

}

//Feed hidden layer into output layer and call activation function

for (int i = 0; i < n\_output; i++) {

for (int j = 0; j < n\_hidden; j++) {

outputLayer.neurons[i]->values[j] = hiddenLayer.neurons[j]->output;

}

outputLayer.neurons[i]->activation();

}

calculateCost(labels);

backpropagate();

}

void Network::classify(std::ifstream& labels) {

labels >> correctOutputs[0] >> correctOutputs[1] >> correctOutputs[2] >> correctOutputs[3] >> correctOutputs[4];

int trueClass;

for (size\_t i = 0; i < correctOutputs.size(); i++) {

if (correctOutputs[i] == 1) {

trueClass = i;

}

}

int outputClass;

float max = 0.0f;

for (size\_t i = 0; i < outputLayer.neurons.size(); i++) {

if (outputLayer.neurons[i]->output > max) {

max = outputLayer.neurons[i]->output;

outputClass = i;

}

}

//Put this classification in the confusion matrix

confusionMatrix[trueClass][outputClass]++;

}

void Network::test(FILE\* image, std::ifstream& labels) {

read(image);

//Call activation function

for (int i = 0; i < n\_hidden; i++) {

hiddenLayer.neurons[i]->activation();

}

//Feed hidden layer into output layer and call activation function

for (int i = 0; i < n\_output; i++) {

for (int j = 0; j < n\_hidden; j++) {

outputLayer.neurons[i]->values[j] = hiddenLayer.neurons[j]->output;

}

outputLayer.neurons[i]->activation();

}

//Check if network classifies image correctly

classify(labels);

}

void Network::printResults(int test\_examples, int hidden) {

//Print the confusion matrix

outfile << " " << "0 " << "1 " << "2 " << "3 " << "4 " << endl;

for (size\_t i = 0; i < confusionMatrix.size(); i++) {

outfile << i << " ";

for (size\_t j = 0; j < confusionMatrix[i].size(); j++) {

if (confusionMatrix[i][j] < 10) {

outfile << confusionMatrix[i][j] << " ";

}

else if (confusionMatrix[i][j] < 100) {

outfile << confusionMatrix[i][j] << " ";

}

else {

outfile << confusionMatrix[i][j] << " ";

}

}

outfile << endl;

}

//Print accuracy

int accuracy = (int)( ((float)(confusionMatrix[0][0] + confusionMatrix[1][1] + confusionMatrix[2][2] + confusionMatrix[3][3] + confusionMatrix[4][4]) / (float)test\_examples) \* 100);

outfile << endl << "The overall accuracy is: " << accuracy << "%" << endl;

outfile << endl << hidden << " neurons were used in the hidden layer." << endl;

outfile << trained\_examples << " images were used to train this neural network." << endl;

outfile << test\_examples << " images were used to test his neural network." << endl;

}

}