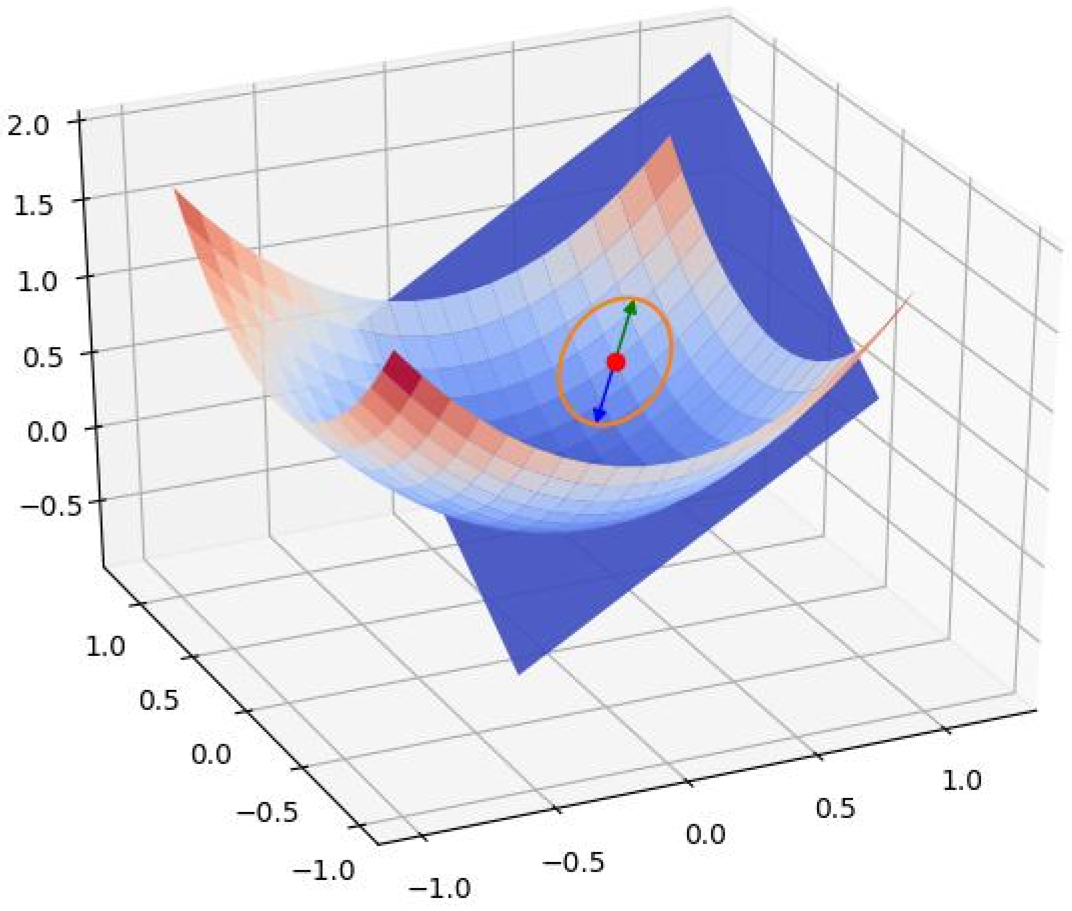
Building A Neural Network From Scratch

C++



Name of Student: Niranjan

Class: XII

Year: 2019 - 2020

Certificate

This is to certify that Niranjan, a student of class XII has successfully completed the research under the guidance of Mrs. Rachna during the year 2019 - 2020 regarding his project titled "*Building a Neural Network From Scratch C++*"

---------------   ---------------

Internal Examiner's Signature   Principal's Signature

External Examiner's Signature

Acknowledgement

I would like to thank my school for giving me the opportunity and for providing all facilities to meet my project requirements.

I wish to express my sincere gratitude to Mrs. Reshma Ganesh, Principal of SSRVM Bangalore East, for the successful outcome of this project.

I would like to thank my computer science teacher Mrs. Rachna for her guidance and support.

I would also like to thank my parents for providing me with the necessary resources to complete this project.

Index

Objective of the Project

The objective of this project is to build a good understanding behind neural networks, by making one from scratch in C++ using no external libraries and a bare minimum of libraries or methods which has not been thought to us in the XI and XII standard of CBSE.

I have used the MNIST handwritten digits data-set to train the neural network.

I have shared all that I have learned by doing this project and hope I do it justice.

Introduction

The project consists of five total files, whose functions have been described:

1. MNISTRead.cpp: Reads the MNIST data and provides it to the rest of the files.

2. DataHandle.cpp: All the File I/O takes place in this file, used to save and load a trained neural network.

3. Network.cpp: The backbone of the project, it has classes that make and train the network.

4. Train.cpp: Uses the Network class from Network.cpp to create, train and save a neural network.

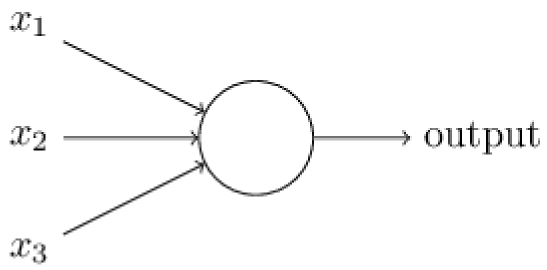
5. runFromModel.cpp: Uses the saved model of a network created in Train.cpp to predict the handwritten digit.

I would like to take this section to describe what the network actually does. I feel it belongs in the introduction as it is not any C++ code

What is a Neural Network?

The Sigmoid

A neural network is a network of “Neurons” that are connected to one another to simulate complex logic and abstract decisions that would be hard to create a hard and fast algorithm for. At the cornet stone of neural networks, we have the *sigmoid neuron*:



This neuron is the building block of the network. It has many inputs and one output, all of them range from [0 - 1]. Each input is associated with a weight ‘W_i’ is the weight associated to the ‘i^{th}’ input of the neuron. Each neuron also has another value, called bias or ‘*B*’. The output of the neuron is calculated as follows:

*Here, a_i refers to the i^{th} input to the neuron

$$output = (\sum_{i=1}^{n} w_i a_i) + b$$

What’s Really Happening Here?

Neural Network as a function

I lied, the sigmoid neuron isn’t that simple, as you may have noticed I mentioned the output lies between [0 - 1], but you might say what if the sum of all the inputs crossed 1?
We fix that by passing the output of the calculation through a special function called.... ‘*sigmoid*’ function! It is defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

*graph on the left
The function outputs values close to 1 for large values of *x* and close to 0 for small values of *x*.

The network is built by many of these neurons connected to each other, each taking input from the previous ones to finally reduce to desired number of outputs, this can hence be treated as a function with many inputs, such as:

$$f(x_1, x_2, x_3, ..., x_n) \rightarrow out_1, out_2, ..., out_m$$

However, since it is so heavily abstracted, we rarely know what the function is doing and just optimize the function as much as we can, but how do we optimize a function whose definition is unknown to us?

The Heart of ‘Machine Learning’

Gradient Descent

This is where gradient descent comes into action. It is a powerful tool that allows us to optimize any function.

This is where it gets interesting. The inputs to out neural network or function are the binary (0, 1) values of the pixels of the handwritten digits, each image being 28 x 28, meaning our function has 784 inputs, but we only want how likely the number we gave as inputs is a digit from 0 - 9, which means we have 10 outputs(0 - 9). Solving the problem of identifying numbers is as good as optimizing our function to a point where it can classify digits its never seen before.

Now that we know what our function is, what value are we aiming to optimize?
For each set of inputs, we want the output to tell us exactly what number it is. This means if we give it an input of the handwritten number ‘2’ the output ‘out_3’ should be maximum(1) and everything else minimum(0). This gives as a value to optimize which is called as the cost function, which grades how good the network has identified the current set of inputs.
*Here cor_i stands for the correct output(0, 1)

$$cost = \frac{1}{2}\sum_{i=0}^{9}(cor_i - out_i)^2$$

This gives us a way to judge how good a network is by passing all the weights and biases to the *cost* function and making the value of *cost* minimum will give us the result we need.
To minimize the *cost*, we start by assuming the network to have random values and calculate the slope or gradient of the cost function, and update all the weights in the negative direction of the slope.

That might be hard to understand, but it has a very good analogy. If we visualize the network having only two weights, then our input would be any value on the (x, y) plane then the output of the *cost* function would be a single value, which is mapped onto the *z-axis* therefore we can visualize the current setup as some function in 3-Dimensions, initializing our network with random values is as good as just picking a random value on the function curve. Now we can imagine a ball at the current position which is rolling down the slopes to find a nice minimum. We can simulate this by calculating the slope at the current position and changing our position by a small value in the negative direction of the slope. The small value will be called *learning rate* from now on, also represented by η . This is the heart of 'machine learning'. The actual calculus involved in finding the slope is simple, but will require a much longer introduction still. I will try my best to explain what is happening as and when it happens in the code.

OBJECTIVES

1. The main objective of the project is to develop a neural network that identifies handwritten digits

2. To overcome the tedious task of recognizing handwritten digits using our brain

3. To focus on the user’s needs and developing a program to meet those requirements

SYSTEM CONFIGURATION

HARDWARE

- Installed Physical memory(RAM): 4.00 GB
- Processor: Intel Core™ i5
- System Type: x64-Based CPU Architecture
- Hard disk: 1TB

SOFTWARE

- Operating System: Arch Linux
- Compiler: g++
- WPS Writer: Documentation

DESCRIPTION

This C++ program on Neural Network has files:
1. MNISTRead.cpp/h
2. DataHandle.cpp/h
3. Network.cpp/h
4. Train.cpp/h
5. runFromModel.cpp/h

The user can edit the Network.h files in easily understandable places to change the topology of the network.

The user can compile and run Train.cpp with the other files, if an argument is passed with launch, a new savefile will be created, else it will attempt to continue training from a previous savefile.

The user can enter the savefile name. The user can enter the *learning rate* and number of *epochs* they would like to train the network for.

The user can compile and run runFromModel.cpp to test the network against the entire data-set or for specific numbers by passing an argument during launch. They can also inference a user drawn file.

HEADER FILES

1. ‘iostream’ - User I/O

2. ‘math.h’ - To calculate exponents and powers

3. ‘ctime, random’ - To initialize network with Gaussian Distribution of random numbers between 0 and 1

4. ‘fstream, string’ - To create and read from savefile with user specified name

MNISTRead.H

```
#ifndef MNISTRead
#define MNISTRead
#include <fstream>
typedef unsigned char uchar;

class MNISTData {
   int l;
   int reverseInt(int);
   std::ifstream labels;
   std::ifstream images;
   int getMagicNum(std::ifstream&);
   public:
   MNISTData(const char *, const char *);
   int* getLabels(int&);
   uchar** getImages(int&, int&);
   ~MNISTData();
};

#endif
```

MNISTRead.CPP

```
/*
   This class handles the reading of the binary image data and label data
   from the MNIST dataset of handwritten digits.
   Functions are described
*/
#include "MNISTRead.H"

/*
   Parameters: (int)Unreversed integer
   Because we read the binary data from left to right while the integers are
   defined with the leftmost being the highest power, we must reverse whatever
   binary data we read to get the actual integer
*/
```

```cpp
int MNISTData::reverseInt(int i) {
   unsigned char c1, c2, c3, c4;
   c1 = i & 255;
   c2 = (i >> 8) & 255;
   c3 = (i >> 16) & 255;
   c4 = (i >> 24) & 255;
   return((int)c1 << 24) + ((int)c2 << 16) + ((int)c3 << 8) + c4;

}

/*
   Parameters:(ifstream) MNIST label file
   returns the first integer that identifies MNIST datasets
*/
int MNISTData::getMagicNum(std::ifstream& file) {
   int magic = 0;
   file.read((char*) &magic, sizeof(int));
   magic = reverseInt(magic);
   return magic;
}

/*
   Constructor:(string) Path to image file, (string) Path to label file
   Sets up the input streams
*/
MNISTData::MNISTData(const char* images, const char* labels) {
   MNISTData::labels.open(labels, std::ios::binary);
   MNISTData::images.open(images, std::ios::binary);
}

//Destructor
MNISTData::~MNISTData() {
   images.close();
   labels.close();
}

/*
   Parameters:(int) Number of images, (int) Image size
   Reads from the images data, and stores the number of images and size
   of image int the parameter variables. It returns a 2D array of unsigned chars
   that have the images. An image as an array of unsigned char that is either binary
'0'
   or binary '1' depending in if the pixel is white or dark.
*/
```

```
uchar** MNISTData::getImages(int& num_images, int& img_size) {
    if(images.is_open()) {
        if(getMagicNum(images) != 2051) std::runtime_error("INVALID IMAGES FILE");
        int rows, col;
        images.read((char*)&num_images, sizeof(int)), num_images = 
reverseInt(num_images);
        images.read((char*)&rows, sizeof(int)), rows = reverseInt(rows);
        images.read((char*)&col, sizeof(int)), col = reverseInt(col);
        img_size = rows * col;
        uchar** _data = new uchar*[num_images];
        for(l = 0; l < num_images; l++) {
            _data[l] = new uchar[img_size];
            images.read((char*)_data[l], img_size);
        }
        return _data;
    } else {
        std::runtime_error("Cant open file images");
        return NULL;
    }
}

/*
    Parameters:(int) Number of labels
    The function reads the label data and returns a unsigned char array of binary 
integers,
    i.e. they have to be cast to int before use
*/
int* MNISTData::getLabels(int& num_labels) {
    if(labels.is_open()) {
        if(getMagicNum(labels) != 2049) std::runtime_error("INVALID LABELS FILE");
        labels.read((char*) &num_labels, sizeof(num_labels)), num_labels = 
reverseInt(num_labels);
        int* _daa = new int[num_labels];
        for(l = 0; l < num_labels; l++) {
            labels.read((char*)&_daa[l], 1);
        }
        return _daa;
    } else {
        std::runtime_error("Cant open file labels");
        return NULL;
    }
}
```

DataHandle.H

```cpp
#ifndef DataHandle
#define DataHandle
#include <fstream>
#include "Network.H"
#include <string.h>
class SaveFile {
   std::fstream fileStream;
   char fname[40];
   int i, j, k;
   void writeWeights(float*, int);
   void check(float);
public:
   SaveFile(const char*);
   void clearSave();
   void writeToFile(Network &n);
   void readToNetwork(Network &n);
};
#endif
```

DataHandle.CPP

```cpp
/*
   This class handles the saving and reading of network weights and biases from
a savefile.
   The class SaveFile has 3 member functions and 1 constructor
   The work of each function is described above the function.
*/

#include "DataHandle.H"
#include <iostream>

#define prnt(x) std::cout << x << '\n'

/*
   Constructor: (string) Name of savefile
   It takes a string literal as an argument and saves the string as the name
   of the file used in all the proceeding operations.
```

```
*/
SaveFile::SaveFile(const char* name) {
    strcpy(fname, name);
}

/*
    Effectively truncates/empties the savefile
*/
void SaveFile::clearSave() {
    fileStream.open(fname, std::ios::trunc|std::ios::out);
    fileStream.close();
}

/*
    Parameters: (Network) The network whose data to save
    Writes all weights and biases of the current network to specified save
*/
void SaveFile::writeToFile(Network &network) {
    fileStream.open(fname, std::ios::out|std::ios::binary);
    float temp = 0;
    int k;
    for(i = 1; i <= Network::num_layers; i++) {
        vector* weights = network.getWeights(i);
        vector biases = network.getBiases(i);
        for(j = 0; j < network.getSizeOfLayer(i); j++) {
            for(k = 0; k < network.getSizeOfLayer(i - 1); k++) {
                temp = *weights[j].of(k);
                fileStream.write((char*)&temp, sizeof(temp));
            }
            temp = *biases.of(j);
            fileStream.write((char*)&temp, sizeof(temp));
        }
    }
    fileStream.close();
}

/*
    Parameters: (Network) The network to read the weights and biases to
    It read the weights and biases from the savefile and sets it to network
    MAKE SURE THE TOPOLOGY IS MATCHING
*/
void SaveFile::readToNetwork(Network &n) {
    fileStream.open(fname, std::ios::in|std::ios::binary);
    float temp;
    int k;
    for(i = 1; i <= Network::num_layers; i++) {
        vector* weights = new vector[n.getSizeOfLayer(i)];
        vector biases(n.getSizeOfLayer(i));
        for(j = 0; j < n.getSizeOfLayer(i); j++) {
            weights[j].make(n.getSizeOfLayer(i - 1));
            for(k = 0; k < n.getSizeOfLayer(i - 1); k++) {
                fileStream.read((char*)&temp, sizeof(temp));
                *weights[j].of(k) = temp;
            }
            fileStream.read((char*)&temp, sizeof(temp));
            *biases.of(j) = temp;
        }
        n.setLayer(i, weights, biases);
    }
    fileStream.close();
}
```

Network.H

```
/*
    Has a vector class that is required to store an array of floats, this makes it
    easier to debug, program, and do calculations
*/
#ifndef Netork
#define Netork
#include <iostream>
#include <math.h>

typedef unsigned char uchar;

/*
    The sigmoid neuron out network is built on
*/
struct neuron {
    float* weight;
    float bias;
    float activation;
};

class vector {
    int size, i, j;
    float* values;
public:
    vector();
    vector(int);
    void display();
    void make(int);
    void set(float val);
    void make(int, float);
    int getSize();
    float* of(int n);
    void add(vector);
    vector* hadamard(vector);//calculates hadamard product
    void divide(float n);
    void destroy();
};


/*
    num_layers and sizes are the editable variables.
*/
class Network {
public:
    //Set this to number of layers, inluding output layer, discluding input layer
    static const int num_layers = 6;
private:
    int i, j;
    uchar* inputLayer = new uchar[784];
    neuron* layers[num_layers];
    vector zVals[num_layers];
    //Set sizes of layers here, last size must always be 10 and first 784, inbetween
can be changed
    int sizes[num_layers + 1] = {784, 128, 64, 128, 64, 128, 10};
    float sigmoid(float x);
    float sigmoidPrime(float x);
public:
    Network();
    void backpropagate(vector, vector*);
    void setInputs(uchar*);
    void setLayer(int layer, vector* w, vector b);
    float findCost(vector expected);
    void findOuts();
    vector* getWeights(int);
    vector getBiases(int);
    vector getActivation(int layer);
    void displayOut();
    void updateBiases(vector averageDels[3]);
    void updateWeights(vector* averageWeightDel[3]);
    int getMaxActivation();
    int getSizeOfLayer(int);
    int getNumLayers();
    ~Network();
};
#endif
```

Network.CPP

```
#define prnt(x) std::cout << x << '\n'
#include "Network.H"

//Define members of vector class. A vector is basically an array of floats.

//Default constructor
vector::vector() {
    size = 0;
}

/*
    Constructor: (int) Length of vector
    Creates a new vector with a specified size.
*/
vector::vector(int n) {
    size = n;
    values = new float[size];
}

/*
    Parameters: (int) Length of vector
    Creates a new vector with a specified size.
*/
void vector::make(int n) {
    if(size != 0) {
        size = n;
        return;
    }
    size = n;
    values = new float[size];
```

```
}

/*
   Parameters: (int) Length of vector, (float) default values of array.
   Creates a new vector with a specified size and sets each element to the default
value.
*/
void vector::make(int n, float def) {
   if(size != 0) {
      delete[] values;
   }
   size = n;
   values = new float[size];
   for(i = 0; i < size; i++) {
      values[i] = def;
   }
}

/*
   Parameters: (int) Value
   Sets all values of vector to the parameter
*/
void vector::set(float val) {
   for(i = 0; i < size; i++) {
      values[i] = val;
   }
}

/*
   Parameters: (int) Value
   Divides all values of vector by the parameter
*/
void vector::divide(float n) {
   for(i = 0; i < size; i++) {
      values[i] /= n;
   }
}

/*
   Parameters: (int) Index
   Returns a pointer to an element of the vector
*/
float* vector::of(int n) {
   return &(values[n]);
```

```
}

/*
   Parameters: (vector) Vector to add
   Adds all the values of vector in parameter to the current vector
*/
void vector::add(vector b) {
   for(i = 0; i < size; i++) {
      values[i] += *b.of(i);
   }
}

/*
   Returns length of vector
*/
int vector::getSize() {
   return size;
}

/*
   Display all values of vector like so [x0, x1, x2, ..., xn]
*/
void vector::display() {
   std::cout << '[';
   for (i = 0; i < size; i++)
   {
      std::cout << values[i] << ',';
   }
   std::cout << "]\n";
}

/*
   Frees memory of vector
*/
void vector::destroy() {
   size = 0;
   delete[] values;
   values = NULL;
}

/*
   Display the activation of the final layer of neurons
*/
void Network::displayOut() {
    std::cout << '[';
    for (i = 0; i < sizes[num_layers]; i++)
    {
        std::cout << layers[num_layers - 1][i].activation << ',';
    }
    std::cout << "]\n";

}

/*
    Network constructor, allocates all the vectors required
*/
Network::Network() {
    for(i = 1; i <= num_layers; i++) {
        layers[i - 1] = new neuron[sizes[i]];
        for(j = 0; j < sizes[i]; j++) {
            layers[i - 1][j].weight = new float[sizes[i - 1]];
        }
        zVals[i - 1].make(sizes[i]);
    }
}

/*
    Parameters: (float) x
    Calculates sigmoid function of 'x'
*/
float Network::sigmoid(float x) {
    float xi = 1.0 / (1 + exp(-x));
    if(xi == 1) {
        return float(0.999);
    } else if(xi == 0) {
        return float(0.001);
    } else {
        return xi;
    }
}

/*
    Feeds the input forward through the network, while saving the activation values
    of each neuron before putting it through the sigmoid - this value is stored in
zVals
*/
void Network::findOuts() {
    float sum = 0;
    int k;
    for(i = 1; i <= num_layers; i++) {
        bool check = i == 1;
        for(j = 0; j < sizes[i]; j++) {
            for(k = 0; k < sizes[i - 1]; k++) {
                sum += layers[i - 1][j].weight[k] * (check ? inputLayer[k] ? 1 : 0 :
layers[i - 2][k].activation);
            }
            sum += layers[i - 1][j].bias;
            *(zVals[i - 1].of(j)) = sum;
            layers[i - 1][j].activation = sigmoid(sum);
            sum = 0;
        }
    }

}

/*
    Parameters: (vector) Vector of the expected activations of the final layer
    Uses the quadratic cost function to find how good/bad the network performed
*/
float Network::findCost(vector expected) {
    float sum = 0;
    for (i = 0; i < sizes[num_layers]; i++) {
        sum += 0.5 * pow(layers[num_layers - 1][i].activation - *expected.of(i), 2);
    }
    return sum;
}

/*
    Parameters: (float) x
    Calculates the derivative of the sigmoid function, evalvuated at x
*/
float Network::sigmoidPrime(float x) {
    float fl = sigmoid(x);
    return fl * (1 - fl);
}

/*
    Parameters: (vector, vector*) First parameter is the expected values of
activation of final layer
    Second parameter is an array of vectors the size is same as number of layers.
Each vector stores the errors of neurons of a layer.
    The error in the final layer's 'ith' neuron is calculated by:
        del(i) = [activation(i) - correct_value(i)] * sigmoid_derivative(activation
of ith neuron before sigmoid or z(i))
    The error in neurons of subsequent layer is calculated by(for the jth neuron
of kth layer):
        del(j, k) = (summation[error in neuron of (k + 1)th layer * weight of the
connection between neurons]) * sigmoid_derivative(z(j))
    This function overwrites all values in the dels(second paramater) and hence doesnt
return anything. In Train.cpp, these values
    are summed and averaged and corrections to the network applied accordingly.
*/
void Network::backpropagate(vector expected, vector* dels) {
    for(i = 0; i < sizes[num_layers]; i++) {
        *dels[num_layers - 1].of(i) = (layers[num_layers - 1][i].activation -
*expected.of(i)) * sigmoidPrime(*(zVals[num_layers - 1].of(i)));
    }
    float sum = 0;
    for(int k = num_layers; k > 1; k--) {
        int prev = k - 1;
        for(i = 0; i < sizes[prev]; i++) { //For each neuron in previous layer
            for(j = 0; j < sizes[k]; j++)  { //For each neuron in current layer
                sum += *dels[k - 1].of(j) * layers[k - 1][j].weight[i];
            }
            sum *= sigmoidPrime(*zVals[prev - 1].of(i));
            *dels[prev - 1].of(i) = sum;
            sum = 0;
        }
    }
}

/*
    Parameters: (vector*) array of vectors consisting of corrections to the bias
    This function updates every bias in the network with the given biases in parameter
*/
void Network::updateBiases(vector averageDels[]) {
    int k;
    for(k = 1; k <= num_layers; k++) {
        for(i = 0; i < sizes[k]; i++) {
            layers[k - 1][i].bias -= *averageDels[k - 1].of(i);
        }
    }
}

/*
```

```
    Parameters: (vector**) array of vectors consisting of corrections to the weights
of every connection between neurons
    This function iterates over every connection, updating the weights of the
connection by the respective value in
    parameter
*/
void Network::updateWeights(vector* averageWeightDel[]) {
    int k;
    for(k = 1; k <= num_layers; k++) {
        for(i = 0; i < sizes[k]; i++) {
            for(j = 0; j < sizes[k - 1]; j++) {
                layers[k - 1][i].weight[j] -= *averageWeightDel[k - 1][i].of(j);
            }
        }
    }
}

/*
    Parameter: (uchar*) Array of unsigned chars which represent the current input
to the network
    The images from MNISTData are read in binary and inputs are set to the same
*/
void Network::setInputs(uchar* ins) {
    for(i = 0; i < 784; i++) {
        inputLayer[i] = ins[i];
    }
}

/*
    Parameters: (int) Layer number
    Returns a vector with all activations of the layer requested
*/
vector Network::getActivation(int layer) {
    vector to_return(sizes[layer]);
    int q;
    if(layer == 0) {
        for(q = 0; q < sizes[layer]; q++) {
            *to_return.of(q) = inputLayer[q] ? 1 : 0;
        }
    } else {
        for(q = 0; q < sizes[layer]; q++) {
            *to_return.of(q) = layers[layer - 1][q].activation;
        }
    }
    return to_return;
}

/*
    Parameters: (int, vector*, vector) Accepts layer number, array of vectors
containing weights of connections of neurons
    in the specified layer and vector containing biases of each neuron in layer
    Sets the data of specified layer with parameters given
*/
void Network::setLayer(int layer, vector w[], vector b) {
    for(i = 0; i < sizes[layer]; i++) {
        for(j = 0; j < sizes[layer - 1]; j++) {
            layers[layer - 1][i].weight[j] = *(w[i].of(j));
        }
        layers[layer - 1][i].bias = *b.of(i);
    }
}

/*
    Returns an integer which is the number which had maximum activation in output
layer
*/
int Network::getMaxActivation() {
    int max = -1;
    float grt = 0;
    for(i = 0; i < sizes[num_layers]; i++) {
        if(layers[num_layers - 1][i].activation > grt) {
            grt = layers[num_layers - 1][i].activation;
            max = i;
        }
    }
    return max;
}

/*
    Parameters: (int) Layer number
    Returns integer which is size of specified layer
*/
int Network::getSizeOfLayer(int n) {
    return sizes[n];
}

/*
```

```
   Parameters: (int) Layer number
   Returns an array of vectors, each containing the weights of all connections of
a neuron in the layer to the previous layer
*/
vector* Network::getWeights(int layer) {
   vector* to_return = new vector[sizes[layer]];
   for(i = 0; i < sizes[layer]; i++) {
      to_return[i].make(sizes[layer - 1]);
      for(j = 0; j < sizes[layer - 1]; j++) {
         *to_return[i].of(j) = layers[layer - 1][i].weight[j];
      }
   }
   return to_return;
}

/*
   Parameters: (int) Layer number
   Returns a vector containing the biases of each neuron in the specifed layer
*/
vector Network::getBiases(int layer) {
   vector to_return(sizes[layer]);
   for(i = 0; i < sizes[layer]; i++) {
      *to_return.of(i) = layers[layer - 1][i].bias;
   }
   return to_return;
}

/*
   Destructor for the network, it frees up all the used memory
*/
Network::~Network() {
   delete[] inputLayer;
   for(i = 1; i <= num_layers; i++) {
      for(j = 0; j < sizes[i]; j++) {
         delete[] layers[i - 1][j].weight;
      }
      delete[] layers[i - 1];
      zVals[i - 1].destroy();
   }
}
```

Train.CPP

```
#include "DataHandle.H"
#include <iostream>
#include <math.h>
#include <ctime>
#include <random>
#include "MNISTRead.H"

#define prnt(x) std::cout << x << '\n'
#define num_layers Network::num_layers

int i, j;

/*
   Shuffles data to make it better minimization
*/
void shuffleImagesAndLabels(uchar** &images, int* &labels, int size) {
   uchar* swap1;
   int swap2;
   int k = size / 2;
   for(int p = 0; p < k; p++) {
      int t = (rand() % k) + 1;
      swap1 = images[i];
      swap2 = labels[i];
      images[i] = images[size - t];
      labels[i] = labels[size - t];
      labels[size - t] = swap2;
      images[size - t] = swap1;
   }
}

/*
   Displays a pretty number!
*/
void display(uchar number[784]) {
   for(int i1 = 0; i1 < 28; i1++) {
      for(int j1 = 0; j1 < 28; j1++) {
         std::cout << (number[i1*28 + j1] ? 'o' : '.') << ' ';
      }
      std::cout << '\n';
   }
```

```
}

/*
    Sets input to the network and displays result
*/
void test(int sampleNumber, Network &network, uchar** &images, int* &label) {
    display(images[sampleNumber]);
    prnt((int)label[sampleNumber]);
    network.setInputs(images[sampleNumber]);
    network.findOuts();
    network.displayOut();
}

int main(int argc, char* argv[]) {
    //Sets the seed for the random generator for shuffling
    srand(time(0));

    //Number of epochs to run for
    int epchs;
    //Learning rate
    float alpha;
    prnt("Enter epochs: ");
    std::cin >> epchs;
    prnt("Enter learning rate: ");
    std::cin >> alpha;

    //Setting up the MNIST data-set
    MNISTData data("Images/images-ubyte", "Images/labels-ubyte");
    int num_img, size;
    uchar** images = data.getImages(num_img, size);
    int* label = data.getLabels(size);

    //Settign up the random generator for Gaussian distribuition
    std::default_random_engine generator;
    std::normal_distribution<float> distribuition(0.0f, 1.0f);

    //Accepting savefile name from user and creating SaveFile object
    char* fname = new char[100];
    prnt("File Name:");
    std::cin >> fname;
    SaveFile save(fname);
    delete[] fname;
    //Checking if the user has entered an argument while execution to cler the save
    if(argc == 2) {
        prnt("Clearing/Creating save\n");
        save.clearSave();
    }

    Network network;
    //If the user wants to create a new save, generate random numbers and set weights
and biases
    if(argc == 2) {
        int layer_sizes[2];
        for(int i = 1; i <= num_layers; i++) {
        layer_sizes[0] = network.getSizeOfLayer(i - 1);
        layer_sizes[1] = network.getSizeOfLayer(i);
        vector w[layer_sizes[1]], b;
        b.make(layer_sizes[1]);
        for(int k = 0; k < layer_sizes[1]; k++) {
            *b.of(k) = distribuition(generator);
        }
        for(int j = 0; j < layer_sizes[1]; j++) {
            w[j].make(layer_sizes[0]);
            for(int k = 0; k < layer_sizes[0]; k++) {
                *w[j].of(k) = distribuition(generator);
            }
        }
        network.setLayer(i, w, b);
        }
    }
    //If user has not entered argument, continue training from existing savefile
by loading the weights into network
    if(argc < 2) {
        prnt("Continuing training.\n");
        save.readToNetwork(network);
    }

    //Current training sample, iterative variables
    int sampleNumber, l1, l2;
    //Vector storing correct outputs
    vector correctVals;
    correctVals.make(network.getSizeOfLayer(num_layers), 0);

    /*
        Vectors to store the values of weights and errors after backpropagating 8
samples, to average the corrction
    */
    vector averageDelta[num_layers];
    vector* averageWeight[num_layers];
    vector delsTemp[num_layers];

    for(i = 0; i < num_layers; i++) {
        averageDelta[i].make(network.getSizeOfLayer(i + 1), 0);
        delsTemp[i].make(network.getSizeOfLayer(i + 1), 0);
        averageWeight[i] = new vector[network.getSizeOfLayer(i + 1)];
        for(j = 0; j < network.getSizeOfLayer(i + 1); j++) {
            averageWeight[i][j].make(network.getSizeOfLayer(i), 0);
        }
    }

    //Stepsize is the gradient step down, 8 is the mini batch size
    float stepsize = 8 / alpha;
    int x;
    int progress = 0;
    for(int epoch = 0; epoch < epchs; epoch++) {
        prnt("Epoch");
        prnt(epoch);
        progress = 0;
        //Running for 6250 mini batches
        for(i = 0; i < 6250; i++) {
            //Running for 8 samples in a minibatch
            for(j = 0; j < 8; j++) {
                sampleNumber = ((i * 8) + j);
                network.setInputs(images[sampleNumber]);
                network.findOuts();
                //Setting the required value in vector correct values to 1(maximum
output) everything else is 0
                *correctVals.of(label[sampleNumber]) = 1;

                //Passing the delsTemp vector and correctVals to backpropogate the
error through the network
                network.backpropagate(correctVals, delsTemp);

                //Use the error found to add the correction required to averageDelta
and averageWeight
                for(x = 0; x < num_layers; x++) {
                    averageDelta[x].add(delsTemp[x]);

                    vector activations = network.getActivation(x);
                    for(l1 = 0; l1 < network.getSizeOfLayer(x + 1); l1++) {
                        for(l2 = 0; l2 < network.getSizeOfLayer(x); l2++) {
```

```
                        *averageWeight[x][l1].of(l2) += *delsTemp[x].of(l1) * 
(*activations.of(l2));
                    }
                }
            }
            //Reset the correct vals to all 0
            *correctVals.of((int)label[sampleNumber]) = 0;
        }
        //After mini batch is done, average the weights and biases using stepsize
        for(x = 0; x < num_layers; x++) {
            averageDelta[x].divide(stepsize);
            for(l1 = 0; l1 < network.getSizeOfLayer(x + 1); l1++) {
                averageWeight[x][l1].divide(stepsize);

            }
        }
        //Pass the averages corrections to the network to change itself
        network.updateBiases(averageDelta);
        network.updateWeights(averageWeight);

        //Reset average delta and average weight to 0
        for(x = 0; x < num_layers; x++) {
            averageDelta[x].set(0);
            for(l1 = 0; l1 < network.getSizeOfLayer(x + 1); 
averageWeight[x][l1].set(0), l1++);
        }
        //Calculate and display current progress to user
        if(int((i/6250.0) * 100) > progress) {
            progress = (i / 6250.0) * 100;
            std::cout << epoch << " - " << progress << '%' << '\n';
        }
    }
    prnt("Shuffling");
    //Save the network after every epoch
    save.writeToFile(network);
    prnt("Saved!\n");
    shuffleImagesAndLabels(images, label, size);
  }
  prnt('\n');

  prnt("DONE.");

  //Testing a random image from the dataset
  test(17, network, images, label);
```

```
    correctVals.destroy();
    for(x = 0; x < num_layers; x++) {
        averageDelta[x].destroy();
        delsTemp[x].destroy();
        for(i = 0; i < network.getSizeOfLayer(x + 1); i++) {
            averageWeight[x][i].destroy();
        }
    }
    for(i = 0; i < num_img; delete[] images[i], images[i] = NULL, i++);
    // for(i = 0; i < 16; w[i].destroy(), i++);
    delete[] label;
    delete[] images;
    images = NULL;
    label = NULL;
    return 0;
}
```

runFromModel.CPP

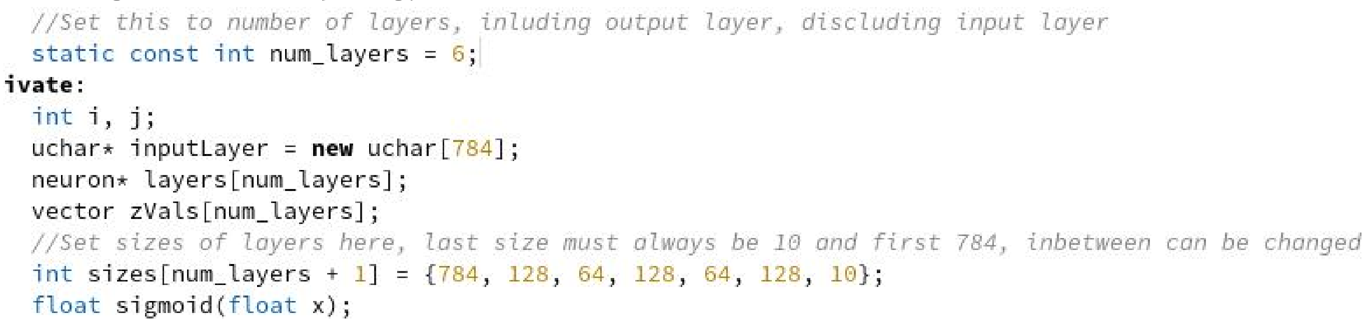
```
#include "DataHandle.H"
#include <iostream>
#include <math.h>
#include "MNISTRead.H"

#define prnt(x) std::cout << x << '\n'

void display(uchar number[784]) {
    for(int i1 = 0; i1 < 28; i1++) {
        for(int j1 = 0; j1 < 28; j1++) {
            std::cout << (number[i1*28 + j1] ? 'o' : '.');
            std::cout << ' ';
        }
        std::cout << '\n';
    }
}

void test(int sampleNumber, Network &network, uchar** &images, uchar* &label) {
    display(images[sampleNumber]);
    prnt((int)label[sampleNumber]);
    network.setInputs(images[sampleNumber]);
    network.findOuts();
    network.displayOut();
```

```
}

int main(int argc, char* argv[]) {
   int i;
   MNISTData data("Images/images-ubyte", "Images/labels-ubyte");
   Network n;


   char* fname = new char[100];
   prnt("File Name:");
   std::cin >> fname;
   SaveFile save(fname);
   uchar** images;
   int* labels;
   int size, num[2];
   images = data.getImages(num[0], size);
   labels = data.getLabels(num[1]);
   save.readToNetwork(n);
   int k = -1;
   //If user argument is -2, inference the network with a raw binary image provided
by user
   if(k == -2) {
      std::ifstream image("Images/img.raw", std::ios::binary);
      uchar ch[784];
      for(int i1 = 0; i1 < 784; i1++) {
         image.read((char*)&ch[i1], 1);
         ch[i1] = !ch[i1];
      }
      n.setInputs(ch);
      n.findOuts();
      n.displayOut();
      prnt(n.getMaxActivation());
      display(ch);
      delete[] labels;
      for(i = 0; i < num[0]; i++) {
         delete[] images[i];
      }
      delete[] images;
      return;
   }
   if(argc > 1) {
      k = int(argv[1][0]) - int('0');
   }
   int correct = 0;
```

```
    int total = 0;
    for(i = 0; i < 60000; i++) {
        if(labels[i] == k || k == -1) {
            total++;
            n.setInputs(images[i]);
            n.findOuts();
            if(int(labels[i]) == n.getMaxActivation()) {
                correct++;
            } else {
                display(images[i]);
                std::cout << "It is: ";
                prnt(int(labels[i]));
                std::cout << "Guessed: ";
                prnt(n.getMaxActivation());
                std::cout << '\n';
            }
            std::cout << "Accuracy- " << float(correct) / (total) << '\n';
        }
    }

    delete[] labels;
    for(i = 0; i < num[0]; i++) {
        delete[] images[i];
    }
    delete[] images;
}
```

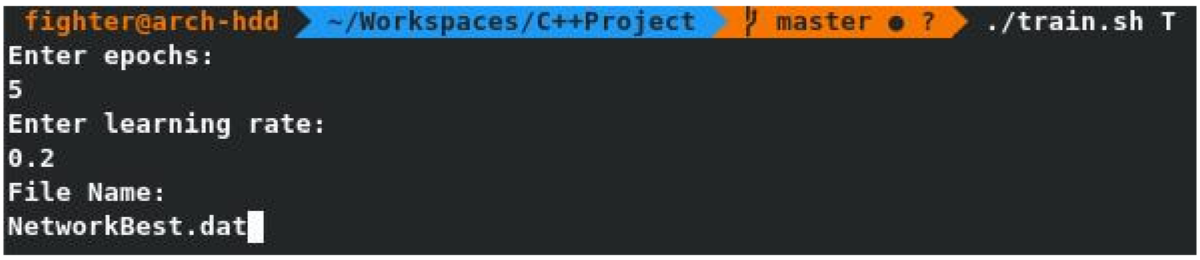
OUTPUT

Editing network topology



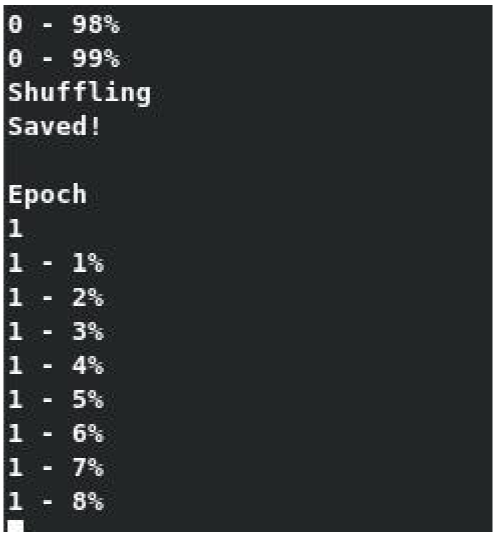
```
  //Set this to number of layers, inluding output layer, discluding input layer
  static const int num_layers = 6;
ivate:
  int i, j;
  uchar* inputLayer = new uchar[784];
  neuron* layers[num_layers];
  vector zVals[num_layers];
  //Set sizes of layers here, last size must always be 10 and first 784, inbetween can be changed
  int sizes[num_layers + 1] = {784, 128, 64, 128, 64, 128, 10};
  float sigmoid(float x);
```

Training Network



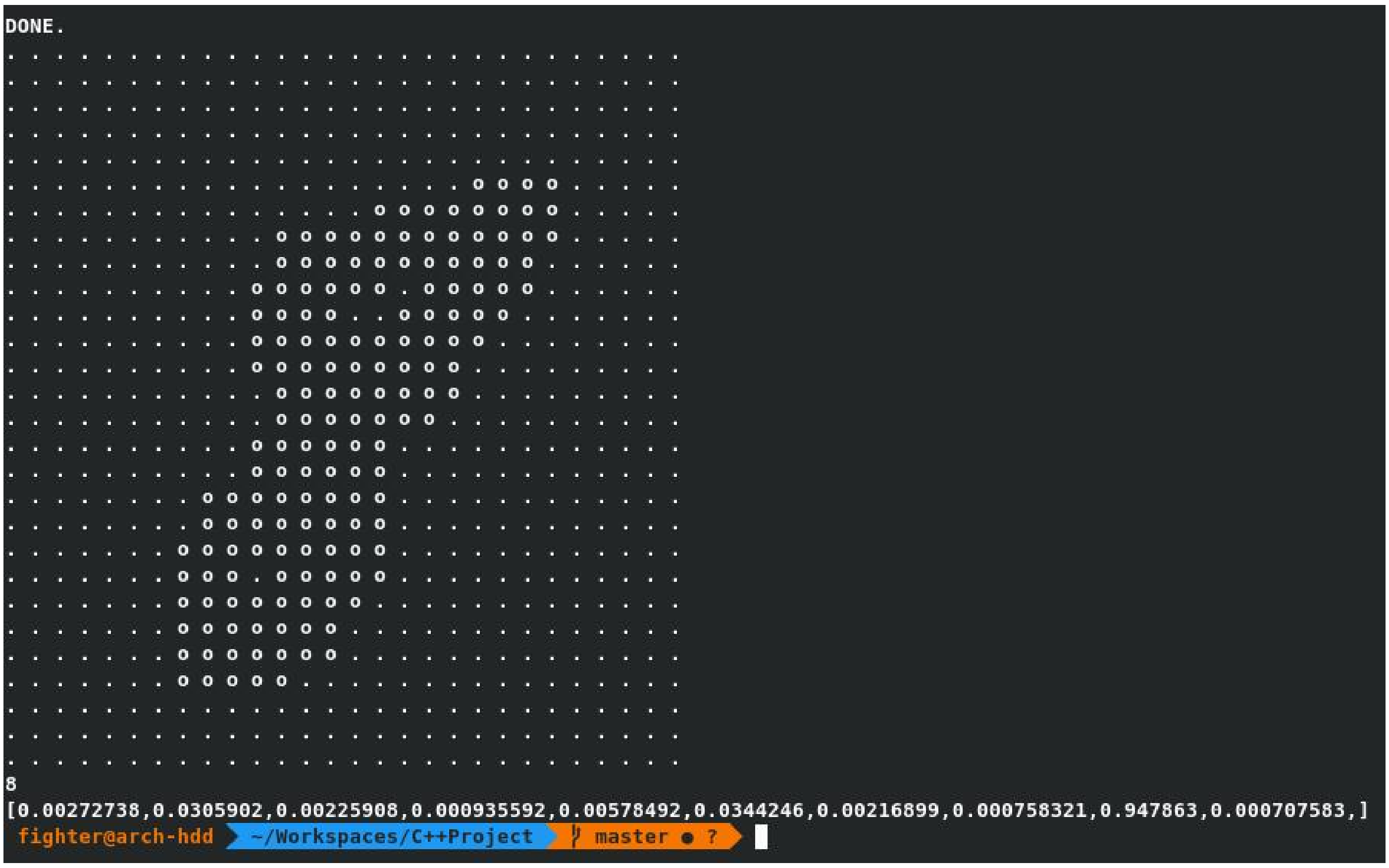
```
 fighter@arch-hdd  ~/Workspaces/C++Project  master ● ?  ./train.sh T
Enter epochs:
5
Enter learning rate:
0.2
File Name:
NetworkBest.dat
```

Training in progress



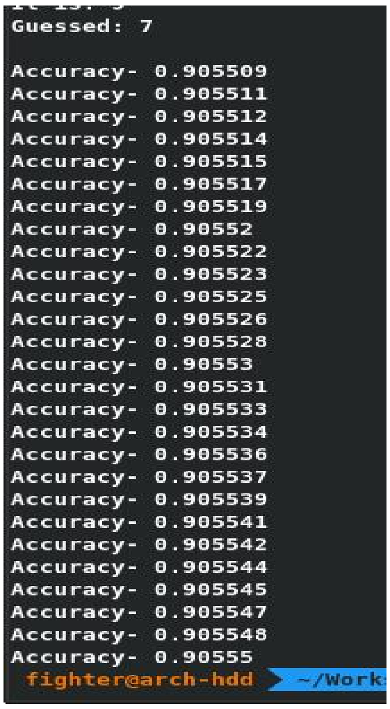
```
0 - 98%
0 - 99%
Shuffling
Saved!

Epoch
1
1 - 1%
1 - 2%
1 - 3%
1 - 4%
1 - 5%
1 - 6%
1 - 7%
1 - 8%
```

Network Completing the epochs(The output is given as 8, and activation of network also displayed)



```
DONE.
. . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . o o o o . . . . .
. . . . . . . . . . . . . . . o o o o o o o o . . . . .
. . . . . . . . . . . o o o o o o o o o o o o . . . . .
. . . . . . . . . . . o o o o o o o o o o o . . . . . .
. . . . . . . . . . o o o o o o . o o o o o . . . . . .
. . . . . . . . . . o o o o . . o o o o o . . . . . . .
. . . . . . . . . . o o o o o o o o o o . . . . . . . .
. . . . . . . . . . o o o o o o o o o . . . . . . . . .
. . . . . . . . . . . o o o o o o o o . . . . . . . . .
. . . . . . . . . . . o o o o o o o . . . . . . . . . .
. . . . . . . . . . o o o o o o . . . . . . . . . . . .
. . . . . . . . . . o o o o o o . . . . . . . . . . . .
. . . . . . . . o o o o o o o o . . . . . . . . . . . .
. . . . . . . . o o o o o o o o . . . . . . . . . . . .
. . . . . . . o o o o o o o o o . . . . . . . . . . . .
. . . . . . . o o o . o o o o o . . . . . . . . . . . .
. . . . . . . o o o o o o o o . . . . . . . . . . . . .
. . . . . . . o o o o o o o . . . . . . . . . . . . . .
. . . . . . . o o o o o o o . . . . . . . . . . . . . .
. . . . . . . o o o o o . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . .
8
[0.00272738,0.0305902,0.00225908,0.000935592,0.00578492,0.0344246,0.00216899,0.000758321,0.947863,0.000707583,]
 fighter@arch-hdd  ~/Workspaces/C++Project  master ● ? 
```

Networks accuracy after only 5 epochs(Tested against whole set)



```
Guessed: 7

Accuracy- 0.905509
Accuracy- 0.905511
Accuracy- 0.905512
Accuracy- 0.905514
Accuracy- 0.905515
Accuracy- 0.905517
Accuracy- 0.905519
Accuracy- 0.90552
Accuracy- 0.905522
Accuracy- 0.905523
Accuracy- 0.905525
Accuracy- 0.905526
Accuracy- 0.905528
Accuracy- 0.90553
Accuracy- 0.905531
Accuracy- 0.905533
Accuracy- 0.905534
Accuracy- 0.905536
Accuracy- 0.905537
Accuracy- 0.905539
Accuracy- 0.905541
Accuracy- 0.905542
Accuracy- 0.905544
Accuracy- 0.905545
Accuracy- 0.905547
Accuracy- 0.905548
Accuracy- 0.90555
 fighter@arch-hdd  ~/Work
```

If you are interested in this project, you can check out the latest version of this project, which will hopefully continue to be updated until I have implemented convolutional neural networks.
Link: https://github.com/LinkG/Project12