Sound Classifier

How Sophisticated Can A Non-Organic Object's Musical Hearing Become Utilizing Neural Networks?

A Computer Science Research Project

By Michael Kuperfish Steinberg

I.D. 214288912

HaKfar HaYarok 2020

Advisor: Yooda Or

Student Contact Information:
Name: Michael Kuperfish Steinberg (מיכאל קופרפיש שטיינברג)
I.D.: 214288912
Date of Birth: 01/06/2003
Address: Oppenheimer 6, Tel Aviv
Tel: +972-58-676-2020
Email: m.kuper.steinberg@gmail.com

General Information:
School: הכפר הירוק ע"ש לוי אשכול
School Tel: 03-645-5666
Field: Computer Science
Study Units: 5 Units

Advisor Contact Information:
Advisor: Yooda Or (יהודה אור)
I.D.: 023098007
Tel: +972-50-734-4457
Email: yoooda@gmail.com
Address: HaKerem 3, Tel Aviv
Academic Degree: MA Engineer, Technion Certified Engineer, Microsoft Certified, Academic transfer to Computer Science on behalf of the country since the year 2000.
Workplaces: Weizmann Institute of Technology, John Bryce College, HaKfar HaYarok, Youth Engineering College of Computer Science.

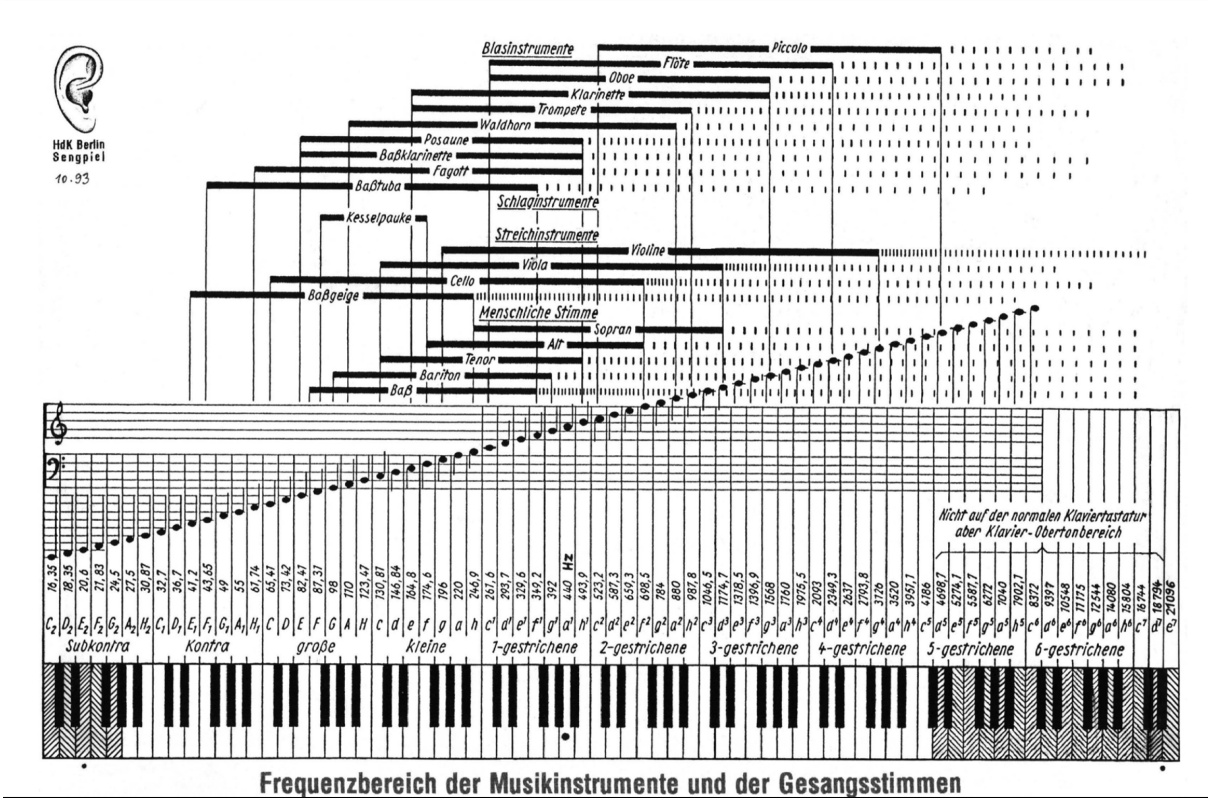
Table of contents:

All code can be found on GitHub: https://github.com/Michael-K-Stein/Sound-Classifier

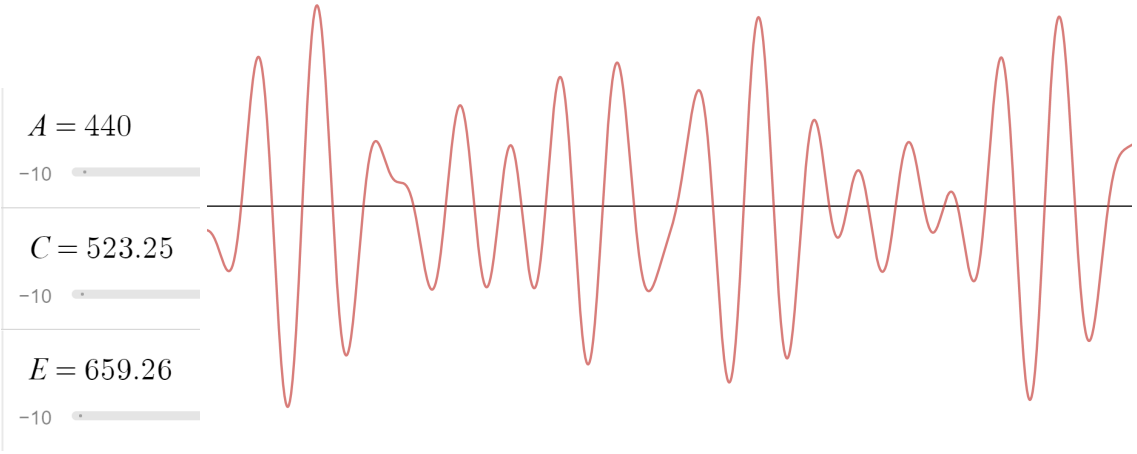
Theoretical Background

Recognizing people by their voice, differentiating between music genres, animal sounds, and understanding the random noises from the kitchen are all standard tasks our human brain does automatically. These are all achieved by our impeccable ability to find patterns. So how do we grant a computer this ability?

The factor humans use to recognize sounds we will be focussing on is timbre. Looking at what "sound" actually is, we see it is a wave moving through space. This wave is generated by vibrations, either from a person's vocal cords, a speaker's diaphragm, or the strings of an instrument. We can now look at Mersenne's formula and see that each chord or string produces a set of frequencies depending on its properties (length, mass per unit length, and the stretching force) $f = \frac{1}{2L}\sqrt{\frac{F}{\mu}}$. Evidently, we can now refer to the sound an instrument makes as a wave at a certain frequency.



Moreover, we can play, for example on a piano, chords. A chord is the simultaneous sound of multiple notes, and thus the sound produced is quite simply a wave of the sum of the individual waves. We can take for example the natural A minor chord, which has the following notes with the corresponding frequencies:



And plot the wave generated by summing the sine waves at these frequencies

$$W(t) = sin(A \cdot t) + sin(C \cdot t) + sin(E \cdot t).$$

This wave describes the A minor chord explicitly (and specifically the A minor chord beginning at the closest A to middle C on a grand piano, A at frequency 440.0 Hz).

Now let's assume we are the audience at this performance, and what we are given is the produced wave (this wave is the movement of particles in the air from the piano's chords, through the room, and into our ears). How could we figure out which notes were played? To do this we use the Fourier Transform. The Fourier Transform takes the wave we "hear" as an input, and returns an array of the frequencies which produce it. In this case, the Fourier Transform will return {440.0, 523.25, 659.26}, and thus tell us exactly which notes were played to produce the sound wave. (This is a very simplified output of the Fourier Transform, a more in-depth explanation will be found below.)

Our brain is trained to do this automatically, and therefore musicians are able to recognize different chords in songs. Additionally, we can also derive information about the theme of the song, if it is uplifting or melancholic, which style it is in, Jazz is a unique example, and much more. This all stems from being able to recognize which frequencies are combined, and utilize them to classify the sound.

It is quite clear to a keen listener that there is a significant difference in the frequency arrays of Twinkle Twinkle Little Star and Seven Nation Army by The White Stripes, while the frequency arrays of Beethoven's 9th Symphony (Ode to Joy) and Vivaldi's Spring will be more alike one-another. This is because different genres of music tend to follow a format, which includes certain frequency ranges, and certain intervals between frequencies. So we can use this fact to teach a computer to differentiate between music genres solely based on the frequency arrays our nifty Fourier Transform provides us with. This is actually how Shazam works (Wang, "An Industrial-Strength Audio Search Algorithm."), and what much of the initial research was based on.

Interestingly, to the best of our understanding, the way the human brain differentiates between peoples' voices is almost identical to how we differentiate between music genres. This means that if we can do one, we can easily do the other.

C++ Research

C++ (or c++, CPP, cpp) is one of the most widely used programming languages. C++ is essentially an extension to the classic C programming language, created by Bjarne Stroustrup. Bjarne Stroustrup developed C++ in 1979 at Bell Labs, as a way to enhance C with more features - most importantly: classes and object oriented programming.

As C++ has classes, let's dive into Object Oriented Programming (OOP):

OOP - Object Oriented Programming

Object Oriented Programming is a method of coding based on objects (classes), rather than Procedural Programming which is based on procedurally calling functions and thus creating routines and subroutines. Classes in OOP contain both code and data, and can interact with other classes, while Procedural Programming would have to pass data pointers between function calls.

The most common examples of Object Oriented Languages are C++, C#, Java, and Python, while Procedural languages include: C, Fortran, COBOL, and Pascal.

Three of the main highlights of OOP are: Encapsulation, Inheritance, and Polymorphism.

Encapsulation

Encapsulation is the concept of combining data with the methods (functions) which use it, and "hiding" that data from unauthorized parties. Data, for example an array of characters (a string), in a class will be accessible only to methods in that class. Therefore, if an external use of the data is made, it is done through a method in the class containing the data, usually through 'get' and 'set' methods. A simple use case of this could be a class "Student" which represents one student in a school, and in this class (disambiguation: programming class. we will refer to the physical room where the students learn as "school") there is a string labeled "Name" which holds the name of the student. We can now use the concept of encapsulation to prevent a students name from being changed (as this should not generally happen) and guarantee that the class returns a properly capitalized name when asked. This class could look like:

```
class Student {
        private:
                char * Name;

        public:
                char error_no_name[9] = "No Name!";
                char * getName() {
```

```
                if (Name == NULL) { return error_no_name; } // Make sure Name
isn't empty
                Name[0] = toupper(Name[0]); // Force capitalize first letter of name
                return Name;
        }

        void setName(int student_name_length, char * student_name) {
                if (Name != NULL) {
                        printf("Student already has a name!\n");
                        return;
                }
                else if (student_name_length < 2) {
                        printf("This cannot be a legal name, it is too short!\n");
                        return;
                }
                else if (student_name == NULL) {
                        printf("No name was given!\n");
                        return;
                }
                else {
                        memcpy(Name, student_name, student_name_length);
                        printf("Student's name was changed to: '%s'.\n", Name);
                }
                return;
        }
};
```

In this example we already see how encapsulation protects data, and places functions in an intuitive location regarding the data they use. In the case we see here, by using encapsulation, we reduce and simplify the code, as we have these protective ‘if’ statements only once, rather than making sure we are not accidentally ruining the student’s name each time we use it.

Inheritance

Inheritance is the mechanism through which one class can be based on, and thus acquire the methods and data of, another class. Inherited classes will have the same attributes and functions as their parent (or base or super) class, while being able to build on those to include more. Inheritance can be used to create a hierarchy of classes, where there is a base class which includes common and standard functionality, and an inherited child class has some extra functionality. For example:

```
typedef struct shape {/* Data */};
typedef struct food {/* Data */ };
typedef struct place {/* Data */ };
```

```
class Animal {
	protected:
		int age;
		shape body;
		food diet;
		place habitat;

	public:
		void Walk() {};
		void Talk();
};
class Bird : public Animal {
	protected:
		int wing_span;
	public:
		void Fly();
};
class Penguin : public Bird {
	public:
		void Swim();
		void Fly() = delete; // Penguins cannot fly
};
```

We have here the hierarchy from a general animal to a penguin, where a penguin has all the attributes and information of a base animal, and also some of the features of a bird - without the flying. Using inheritance, we avoid writing definitions and implementations of standard functions like “Walk()” in each class of animal. Say we implemented a class for each of the ~400 dog breeds. It would be incredibly redundant to implement 400 walk functions, while if we inherited each ‘dog_breed’ class from a base ‘dog’ class, we would only implement this function - and many others - once.

Polymorphism

Polymorphism allows two interesting and useful features which can be classified as static and dynamic polymorphisms. Static polymorphism (or ad hoc polymorphism) allows multiple functions two have the same name, and differentiates between them by their formal parameters - function overloading. This allows us to define similar functions which use different types of parameters with the same name:

```
double Sum(int num1, int num2) {
	return num1 + num2;
}
double Sum(double num1, double num2) {
```

```
        return num1 + num2;
}
double Sum(float num1, float num2) {
        return num1 + num2;
}
double Sum(int length_of_array, double * num_array) {
        double sum = 0;
        for (int i = 0; i < length_of_array; i++) {
                sum += num_array[i];
        }
        return sum;
}
```

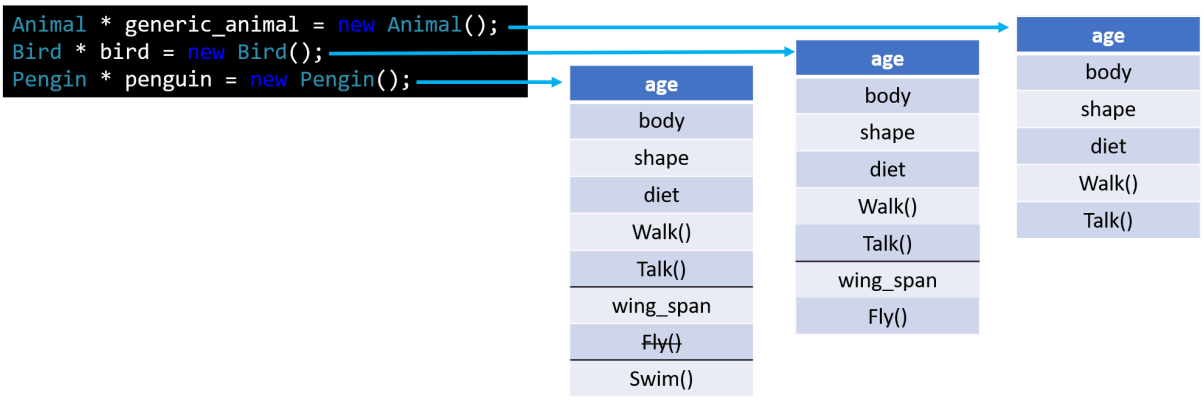
Now when we need to use some implementation of the ‘Sum’ function, we do not need to worry about which one to choose, as the compiler will select the proper one for each call according to that call's parameters.

The second type, dynamic polymorphism, allows us to refer to objects of different derived classes the same way, given the derive from a common function. We can use this feature to iterate over an array of animals (see Encapsulation) and call their common “Walk()” function.

```
Animal * animals[4];

Animal * generic_animal = new Animal();
Bird * bird = new Bird();
Penguin * penguin = new Penguin();
Dog * dog = new Dog(); // assume we implement a class Dog similarly to Penguin.

animals[0] = generic_animal;
animals[1] = bird;
animals[2] = penguin;
animals[3] = dog;

for (int i = 0; i < 4; i++) {
        animals[i]->Walk();
}
```

Here we see that, although each of the four objects in the array are of a different type, we can store them alongside each other, and call their common functions in such a way.

This is granted by how inheritance is implemented in memory, which is that the base class’ attributes and methods are at the beginning and are then followed by the child class’ information.


Animal * generic_animal = new Animal();
Bird * bird = new Bird();
Pengin * penguin = new Pengin();
age
body
shape
diet
Walk()
Talk()
wing_span
Fly()
Swim()
age
body
shape
diet
Walk()
Talk()
age
body
shape
diet
Walk()
Talk()

Evolvement - Practical Research

One of the most significant challenges which rose during this project, was that the platform I was using to run and execute the machine learning algorithms is a standard home computer - with the processing power of one. This means that each iteration over, for example, a euclidean distance calculation function took a massive amount of time, on the order of 1,500 milliseconds. This function being called 4,000 times for each of the 1,200 data points in the prototype-phase dataset, meant training the neural network would take 2,000 hours.

To increase efficiency to a more feasible training time, I tried the following:

1. Calling Conventions
    a. Calling conventions are the standard methods by which functions are called - each convention defines how arguments are passed into the function and how where the output is placed. These conventions are all implemented in low-level languages, in our case we will use Assembly 8086 (x86) - although the actual project is in 64bit assembly.
    b. The standard calling convention is ‘cdecl’ (C declaration), which passes arguments to the function through the stack. This means that code in C like:

```
int func(int a,int b,int c,int d, int e, int f) {
   int out = a + b + c + d + e + f;
   return out;
}

int main(int argc, char ** argv) {
   int output = func(0,1,2,3,4,5);
}
```

Will look like:

```
_func   PROC
        push    ebp
        mov     ebp, esp
        push    ecx
        mov     eax, DWORD PTR [ebp + 8]
        add     eax, DWORD PTR [ebp + 12]
        add     eax, DWORD PTR [ebp + 16]
        add     eax, DWORD PTR [ebp + 20]
        add     eax, DWORD PTR [ebp + 24]
        add     eax, DWORD PTR [ebp + 28]
        mov     DWORD PTR [ebp - 4], eax
        mov     eax, DWORD PTR [ebp - 4]
        mov     esp, ebp
        pop     ebp
        ret     0
```

```
_func   ENDP

_main   PROC
        push    ebp
        mov     ebp, esp
        push    ecx
        push    5
        push    4
        push    3
        push    2
        push    1
        push    0
        call    _func
        add     esp, 24                         ; 00000018H
        mov     DWORD PTR [ebp - 4], eax
        xor     eax, eax
        mov     esp, ebp
        pop     ebp
        ret     0
_main   ENDP
```

While this is a good convention, and for general use is sufficient, in our case we need to optimize the code further, and remove as many redundant instructions as possible. The first thing to do, is change the calling convention to ‘fastcall’ which will pass the arguments through registers instead of the stack. Luckily, we only actually need two arguments, so the function could look like this:

```
func:
        push    rbp
        mov     rbp, rsp
        mov     edx, edi
        mov     eax, esi
        add     eax, edx
        mov     DWORD PTR [rbp-4], eax
        mov     eax, DWORD PTR [rbp-4]
        pop     rbp
        ret
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 32
        mov     DWORD PTR [rbp-20], edi
        mov     QWORD PTR [rbp-32], rsi
        mov     esi, 1
        mov     edi, 0
        call    func
        mov     DWORD PTR [rbp-4], eax
```

```
    mov     eax, 0
    leave
    ret
```

Notice how we reduced the amount of instructions in *func* which will be expressed as a significant speed increase when we repeatedly call the function.

2. Parameter Types

a. In C/C++ parameters can be passed to function either by-value, meaning you pass the literal bits of the value to the function, or by-reference, meaning you pass an address pointer to the value in memory.

b. By-Value is clearly faster, as we avoid having to dereference the address when using the value. However, there is a caveat to by-value, and an exception:

   i. The caveat: An "object" passed by-value cannot be changed. Any changes done to the object are temporary within the scope of the called function, as the function actually received a bitwise copy of the object, and this copy is in a different memory location than the original.

   ii. The exception: "Objects" or structures cannot be properly passed by-value. While some compilers (not MSVC, which was used to compile most of this project) allow passing a structure by-value, this is quite odd and wildly inefficient.

c. By-Reference, while being slower, solves the caveat of being unable to change the object outside the scope of the function.

   i. Notice the redundancy, however, with passing a standard object - like an int32_t or char - by-reference in order to change it:

```
int square(int num) {
   return num * num; // Multiplies num
}
void square(int * num) {
   *num = (*num) * (*num); // Changes num at its original location as allocated in main.
}

int main() {
   int number = 5;
   number = square(number);

   int number2 = 7;
   square(&number);
}
```

```
square(int):
        push    rbp
```

```
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, eax
        pop     rbp
        ret
square(int*):
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-8], rdi
        mov     rax, QWORD PTR [rbp-8]
        mov     edx, DWORD PTR [rax]
        mov     rax, QWORD PTR [rbp-8]
        mov     eax, DWORD PTR [rax]
        imul    edx, eax
        mov     rax, QWORD PTR [rbp-8]
        mov     DWORD PTR [rax], edx
        nop
        pop     rbp
        ret
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-8], 5
        mov     eax, DWORD PTR [rbp-8]
        mov     edi, eax
        call    square(int)
        mov     DWORD PTR [rbp-8], eax
        mov     DWORD PTR [rbp-4], 7
        lea     rax, [rbp-8]
        mov     rdi, rax
        call    square(int*)
        mov     eax, 0
        leave
        ret
```

A very simple 7 line function mutates into a complex 13 line function. This is an easily avoidable redundancy.

Therefore, for performance critical functions, we will pass only simple types by-value.

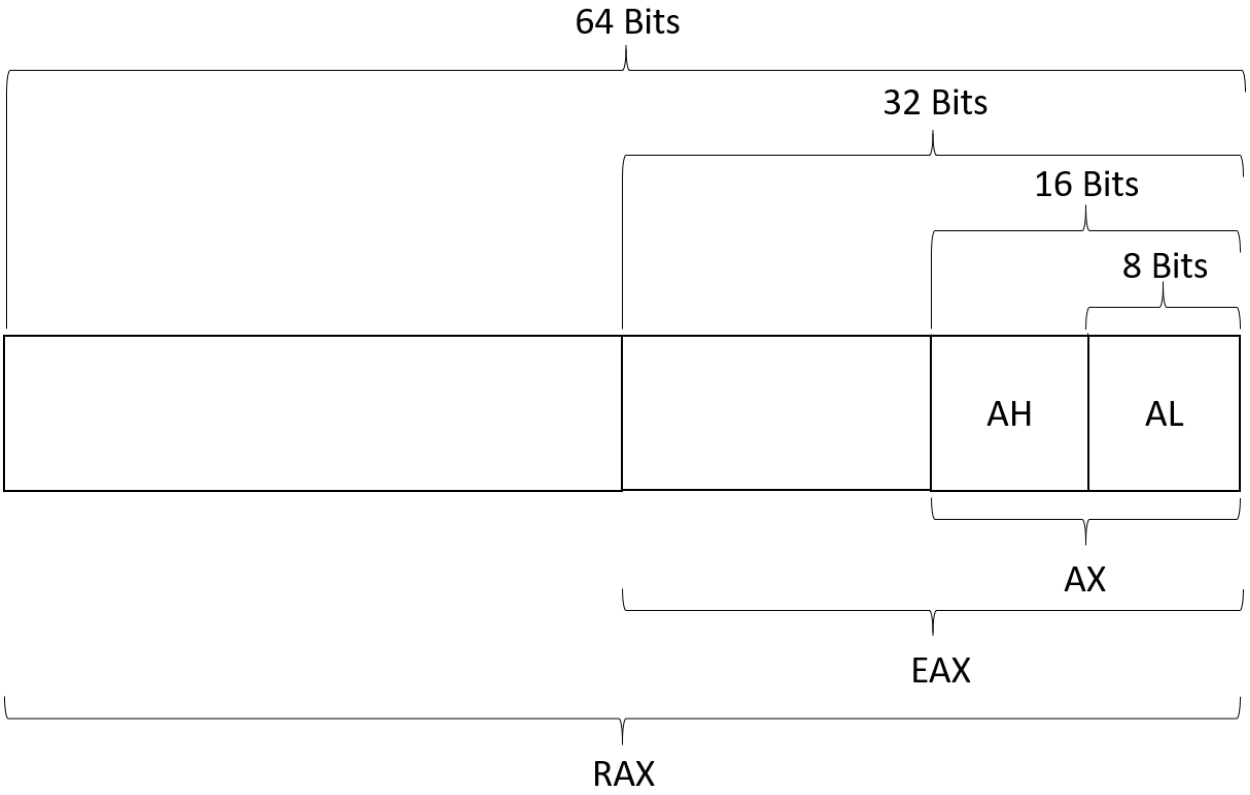
3. Arrays vs Vectors

a. However, even with all these optimizations, most functions, specifically uint32_t knn::calculate_distance(data * query_point, data * input), were still severely hindering the process.

b. The original version of knn:calculate_distance(data*, data*) looked a little like this:

```
double knn::calculate_distance(data * query_point, data * input) {
   double distance = 0.0;
   if (query_point->get_feature_vector_size() !=
input->get_feature_vector_size()) {
      printf("Vector size mismatch");
      exit(1);
   }

   for (uint16_t i = 0; i < query_point->get_feature_vector_size(); i++) {
      distance += pow(query_point->get_feature_vector()->at(i) -
input->get_feature_vector()->at(i),2);
   }
   distance = sqrt(distance);
   return distance;
}
```

c. The first optimization I made, was realizing that the feature vectors were of type std::vector<uint8_t>, meaning that the difference between two values on two vectors was also of type uint8_t (assuming we take the absolute value, which we do). This automatically means we are working with 8 bit integers instead of 64 bit doubles. Using these 8 bit integers, we gain the capability to use 8 bit operations, those using AL, AH, BL, …, DH, which are significantly faster than 64 bit operations. (Note: during the research, I initially tried using floating points (float) instead of doubles, in the hopes that using Floating-Point arithmetic - which can be done on the Floating Point Unit (FPU) - would be faster. Though this was successful and did in-fact increase performance, ultimately it was inferior to simply using unsigned byte-sized integers). One of my reasons for using 8 bit integers was the idea that twice as many could fit in the computer's registers (see below) and thus reduce time spent fetching values from RAM (Random Access Memory) or the Stack.



d. Now to the crux of the issue; Vectors. Although vectors massively simplify dynamic size list usage, they are exponentially slower than normal c-style arrays. The main problem in our case with vectors is the ‘at(int)’ function, which through trial-and-error, I found to be the bottleneck of our distance function. To solve this, I added another step in the initialization of the entire algorithm: after setting everything up, copy each and every vector in every single object found into a corresponding array inside that object. Obviously, this additional step nearly doubles the initialization time. However, we only need to do this step once, and by doing so we reduce the processing time of each data point by a factor of 10. The actual implementation of this was quite simple, as I merely duplicated all the performance critical functions and wrote a “fast” version which used arrays instead of vectors. The final distance function looks like this:

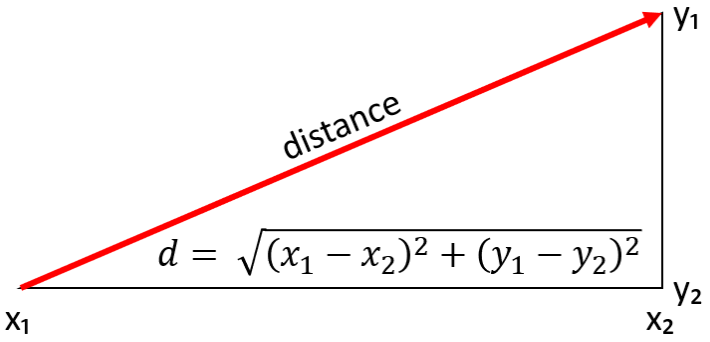
```
uint32_t knn::calculate_distance_fast(data * query_point, data * input) {
    uint32_t distance = 0;

    uint8_t * arr1 = query_point->get_feature_array();
    uint8_t * arr2 = input->get_feature_array();

    uint16_t size = query_point->get_feature_vector_size();

    for (uint16_t i = 0; i < size; i++) {
```

```
        distance += pow(arr1[i] - arr2[i], 2);
    }
    //distance = sqrt(distance); //Not necessary for comparisons

    return distance;
}
```

i. We use integer types of the smallest possible sizes; unsigned 32 bit for the total distance - as it can be quite large, unsigned 8 bit for the inputs - as they are simply those (see our FFT class), and unsigned 16 bit for the size of the arrays (which is obviously the same as the vector's size) - as this is around 4,000 in our case.

ii. We moved the array length validation out of this function, as to not check each and every time we call the distance function, rather only before beginning the distance comparisons.

iii. We do not perform the square root calculation on the total distance. We use euclidean distance, which, informally, looks like this:



And formally:

Assume points a and b such that their coordinates in an N dimensions space are $\{a_i \mid i \in \mathbb{R} \wedge 1 \leq i \leq N\} = \{a_1, a_2, \ldots, a_n\}$ and $\{b_1, b_2, \ldots, b_n\}$.

The euclidean distance between points a and b is defined as:

$$d = \sqrt{\sum_{i=1}^{N} (a_i - b_i)^2}$$

However, if all we care about is the relation between distances, we do not need to square root the sum since:

$\forall a,b \in \mathbb{R}_+.\ a > b \Longleftrightarrow \sqrt{a} > \sqrt{b}$

iv. We do not need to ask the data points' class for the array or vector for each value, as we store our own pointers with 'arr1' and 'arr2'. This

trivial fix reduces the extra calls to the ‘get_feature_array()’ function, which while being very compact, still is redundant.

All these improvements reduced the processing time of the algorithm from a couple thousand hours, to a few minutes. Specifically the bottleneck of knn::find_knearest(data*), which loops over the calculate_distance function many times, improved from around 1,850 milliseconds per loop, to under 330 milliseconds => a 82.16% increase in speed.

Machine Learning

Introduction

Machine learning is a form of artificial intelligence, designed to be able to solve problems which were not anticipated. Artificial intelligence (AI), is the concept wherein a computer can make decisions based on given information, to "think" so to say. The simplest form of AI is the commonly used "if - else" statement, which produces an output depending on the input.

```
if (time == Night) {
    turn_lights_on();
} else {
   turn_lights_off();
}
```

However, there is a clear limitation to "if - else" statements; the conditions must be predefined and the options for variability are very limited. An AI utilizing "if - else" statements could sort a school of children into classes by age, yet would be discomposed given a picture of a student to place in an appropriate class. Machine learning overcomes this limitation by making decisions in a more "organic" fashion, similar to human thought, as, pertaining to the aforementioned example, a school teacher would easily be able to tell the age of a child off a picture.

While theoretically possible, hardcoding a program to recognize a person's age from a picture is both unfathomable and impractical. Computers rely on functions, a mathematical concept which describes the relation or expression between variables ("Function", Vocabulary.com). Humans can intuitively see and understand functions in two dimensions (X - Y Plane / Cartesian Plane), and can grasp a three dimensional coordinate system (X - Y - Z). However, some decisions rely on more than two quantifiable real-number inputs ("input" $\in \mathbb{R}$), and humans simply lack the conceptual ability to turn such a function into code.

Concept and Theory

To solve problems which humans cannot explain, machine learning practically "creates" its own function for the problem. Instead of being told how to solve a problem, a machine learning algorithm is fed problems and their solutions, and depending on the machine learning algorithm, either learns and refines its "function" according to the training data, or uses the training data as a comparison to the validation and test data.

MNIST Database - Handwritten Digits

The MNIST database is a collection of 60,000 grayscale images of handwritten digits in a 28x28px box.

In order to research and learn more about machine learning, I wrote three kinds of machine learning algorithms to classify the digits: KNN, K Means, and a Neural Network.

Some of the code was the same throughout all three algorithms. Hence it is not placed under a specific category, rather in a general location. This includes the classes of the data itself, data holding classes and graphics (Which is solely for debugging purposes in the KNN algorithm). Note: there is a “main” function for each algorithm, only one of them may be available (uncommented) at a time.

Code

data.h + data.cpp

```
#ifndef __DATA_H
#define __DATA_H

#include "stdint.h"
#include "stdio.h"
#include <vector>


#pragma once
class data
{

	std::vector<uint8_t> * feature_vector;
	std::vector<double> * double_feature_vector;
	std::vector<double> *normalizedFeatureVector;
	std::vector<int> * class_vector;
	uint8_t label;
	int enum_label; // A: 1; B: 2
	double distance;

	public:
	data();
	~data();

	void set_feature_vector(std::vector<uint8_t> *);
	void set_double_feature_vector(std::vector<double> *);
	void setNormalizedFeatureVector(std::vector<double>*);
	void set_class_vector(int count);
	void append_to_feature_vector(uint8_t);
	void append_to_feature_vector(double);
	void set_label (uint8_t);
	void set_enumerated_label(int);
	void set_distance(double);

	double get_distance();
	int get_feature_vector_size();
	uint8_t get_label();
```

```
        uint8_t get_enumerated_label();
        std::vector<uint8_t> * get_feature_vector();
        std::vector<double> * get_double_feature_vector();
        std::vector<double> * getNormalizedFeatureVector();
        std::vector<int> * get_class_vector();

};

#endif
```

```
#include "data.h"

data::data()
{
        feature_vector = new std::vector<uint8_t>;
}


data::~data()
{
}

std::vector<double> * data::getNormalizedFeatureVector()
{
        return normalizedFeatureVector;
}
void data::setNormalizedFeatureVector(std::vector<double>* vect)
{
        normalizedFeatureVector = vect;
}

void data::set_feature_vector(std::vector<uint8_t> * vect) {
        feature_vector = vect;
}

void data::set_double_feature_vector(std::vector<double> * vect) {
        double_feature_vector = vect;
}
```

```
void data::set_class_vector(int count) {
	class_vector = new std::vector<int>();
	for (int i = 0; i < count; i++)
	{
		if (i == label)
			class_vector->push_back(1);
		else
			class_vector->push_back(0);
	}
}

void data::append_to_feature_vector(uint8_t val) {
	feature_vector->push_back(val);
}
void data::append_to_feature_vector(double val) {
	normalizedFeatureVector->push_back(val);
}

void data::set_label(uint8_t val) {
	label = val;
}

void data::set_enumerated_label(int val) {
	enum_label = val;
}

int data::get_feature_vector_size() {
	return feature_vector->size();
}

uint8_t data::get_label() {
	return label;
}

uint8_t data::get_enumerated_label() {
	return enum_label;
}

void data::set_distance(double val) {
	distance = val;
}
```

```
std::vector<uint8_t> * data::get_feature_vector() {
	return feature_vector;
}

std::vector<double> * data::get_double_feature_vector() {
	return double_feature_vector;
}

std::vector<int> * data::get_class_vector() {
	return class_vector;
}

double data::get_distance() {
	return distance;
}
```

data_handler.h + data_handler.cpp

```
#ifndef __DATA_HANDLER_H
#define __DATA_HANDLER_H

#include <fstream>
#include "stdint.h"
#include "data.h"
#include <vector>
#include <string>
#include <map>
#include <unordered_set>
#include <algorithm>
#include "Graphic.h"

#pragma once
class data_handler
{

	std::vector<data *> * data_array;
	std::vector<data *> * training_data;
	std::vector<data *> * test_data;
	std::vector<data *> * validation_data;
```

```
        int num_classes;
        int feature_vector_size;
        std::map<uint8_t, int> class_map;
        std::map<std::string, int> classMap;

        const double TRAIN_SET_PERCENT = 0.75;
        const double TEST_SET_PERCENT = 0.20;
        const double VALIDATION_PERCENT = 0.05;

        public:
        data_handler();
        ~data_handler();

        void read_csv(std::string path, std::string delimiter);
        void read_feature_vector(std::string filePath);
        void read_feature_labels(std::string filePath);
        void split_data();
        void count_classes();
        void normalize();

        uint32_t convert_to_little_endian(const unsigned char * bytes);

        int get_class_count();

        std::vector<data *> * get_training_data();
        std::vector<data *> * get_test_data();
        std::vector<data *> * get_validation_data();

        private:
        uint32_t max_image_array = 1000; // To keep the program from hogging the RAM
};

#endif
```

```
#include "data_handler.h"

data_handler::data_handler()
{
```

```
	data_array = new std::vector<data *>;
	test_data = new std::vector<data *>;
	training_data = new std::vector<data *>;
	validation_data = new std::vector<data *>;
}


data_handler::~data_handler()
{
}

void data_handler::read_csv(std::string path, std::string delimiter) {
	num_classes = 0;
	std::ifstream data_file(path.c_str());
	std::string line;
	while (std::getline(data_file, line)) {
		if (line.length() == 0) { continue; }
		data * d = new data();
		d->set_double_feature_vector(new std::vector<double>());
		size_t position = 0;
		std::string token; // value in between delimiter
		while ((position = line.find(delimiter)) != std::string::npos) {
			token = line.substr(0, position);
			d->append_to_feature_vector(std::stod(token));
			line.erase(0, position + delimiter.length());
		}
		if (classMap.find(line) != classMap.end()) {
			d->set_label(classMap[line]);
		}
		else {
			classMap[line] = num_classes;
			d->set_label(classMap[line]);
			num_classes++;
		}
		data_array->push_back(d);
	}
	feature_vector_size = data_array->at(0)->get_double_feature_vector()->size();
}

void data_handler::read_feature_vector(std::string filePath) {
	uint32_t		header[4]; // Magic | Num Images | Row Size | Col Size
```

```
        unsigned char  bytes[4];
        FILE *f = fopen(filePath.c_str(), "rb");
        if (f) {
                for (int i = 0; i < 4; i++) {
                        if (fread(bytes, sizeof(bytes), 1, f)) {
                                header[i] = convert_to_little_endian(bytes);
                        }
                }
                printf("Done getting input file header.\n");
                uint32_t image_size = header[2] * header[3];

                uint32_t image_count = std::min(header[1], max_image_array);

                for (int i = 0; i < image_count; i++)
                {
                        data *d = new data();
                        d->set_feature_vector(new std::vector<uint8_t>());
                        uint8_t element[784];
                        fread(element, 1, image_size, f);
                        for (int j = 0; j < image_size; j++)
                        {
                                if (true) {
                                        d->append_to_feature_vector(element[j]);
                                }
                                else {
                                        printf("Error reading from file.\n");
                                }
                        }

                        data_array->push_back(d);
                }

                normalize();

                printf("Successfully read and stored %lu feature vectors.\n",
data_array->size());
        } else {
                printf("Could not find file.\n");
        }
}
```

```
void data_handler::read_feature_labels(std::string filePath) {
	uint32_t		header[2]; // Magic | Num Images
	unsigned char	bytes[4];
	FILE *f = fopen(filePath.c_str(), "rb");
	if (f) {
		for (int i = 0; i < 2; i++) {
			if (fread(bytes, sizeof(bytes), 1, f)) {
				header[i] = convert_to_little_endian(bytes);
			}
		}
		printf("Done getting label file header.\n");

		uint32_t image_count = std::min(header[1], max_image_array);

		for (int i = 0; i < image_count; i++) {
			data *d = new data();
			uint8_t element[1];

			if (fread(element, sizeof(element), 1, f)) {
				data_array->at(i)->set_label(element[0]);
			}
			else {
				printf("Error reading from file.\n");
				exit(1);
			}
		}
		printf("Successfully read and stored label vectors.\n");
	}
	else {
		printf("Could not find file.\n");
	}
}

void data_handler::normalize()
{
	std::vector<double> mins, maxs;
	// fill min and max lists

	data *d = data_array->at(0);
	for (auto val : *d->get_feature_vector())
	{
```

```
                mins.push_back(val);
                maxs.push_back(val);
        }

        for (int i = 1; i < data_array->size(); i++)
        {
                d = data_array->at(i);
                for (int j = 0; j < d->get_feature_vector_size(); j++)
                {
                        double value = (double)d->get_feature_vector()->at(j);
                        if (value < mins.at(j)) mins[j] = value;
                        if (value > maxs.at(j)) maxs[j] = value;
                }
        }
        // normalize data array

        for (int i = 0; i < data_array->size(); i++)
        {
                data_array->at(i)->setNormalizedFeatureVector(new
std::vector<double>());
                data_array->at(i)->set_class_vector(get_class_count());
                for (int j = 0; j < data_array->at(i)->get_feature_vector_size(); j++)
                {
                        if (maxs[j] - mins[j] == 0)
data_array->at(i)->append_to_feature_vector(0.0);
                        else

data_array->at(i)->append_to_feature_vector((double)(data_array->at(i)->get_feature_vect
or()->at(j) - mins[j]) / (maxs[j] - mins[j]));
                }
        }
}


void data_handler::split_data() {
        std::unordered_set <int> used_indexes;
        int train_size = data_array->size() * TRAIN_SET_PERCENT;
        int test_size = data_array->size() * TEST_SET_PERCENT;
        int valid_size = data_array->size() * VALIDATION_PERCENT;

        // Training Data
```

```
uint32_t count = 0;
while (count < train_size) {
        uint32_t rand_index = count;
        //uint32_t rand_index = rand() % data_array->size();  // Takes too long
        if (used_indexes.find(rand_index) == used_indexes.end()) {
                training_data->push_back(data_array->at(rand_index));
                used_indexes.insert(rand_index);
                count++;
        }
}

// Test Data

count = 0;
while (count < test_size) {
        //uint32_t rand_index = rand() % data_array->size();  // Takes too long
        uint32_t rand_index = train_size + count;
        if (used_indexes.find(rand_index) == used_indexes.end()) {
                test_data->push_back(data_array->at(rand_index));
                used_indexes.insert(rand_index);
                count++;
        }
}

// Validation Data

count = 0;
while (count < valid_size) {
        //uint32_t rand_index = rand() % data_array->size();  // Takes too long
        uint32_t rand_index = train_size + test_size + count;
        if (used_indexes.find(rand_index) == used_indexes.end()) {
                validation_data->push_back(data_array->at(rand_index));
                used_indexes.insert(rand_index);
                count++;
        }
}

printf("Training Data Size: %lu.\n", training_data->size());
printf("Test Data Size: %lu.\n", test_data->size());
printf("Validation Data Size: %lu.\n", validation_data->size());
```

```
}

void data_handler::count_classes() {
	int count = 0;
	for (unsigned i = 0; i < data_array->size(); i++) {
		if (class_map.find(data_array->at(i)->get_label()) == class_map.end()) {
			class_map[data_array->at(i)->get_label()] = count;
			data_array->at(i)->set_enumerated_label(count);
			count++;
		}
	}
	num_classes = count;

	printf("Extracted %d unique classes.\n", num_classes);
}

uint32_t data_handler::convert_to_little_endian(const unsigned char * bytes) {
	return (uint32_t)((bytes[0] << 24) | (bytes[1] << 16) | (bytes[2] << 8) | (bytes[3]));
}

std::vector<data *> * data_handler::get_training_data() {
	return training_data;
}
std::vector<data *> * data_handler::get_test_data() {
	return test_data;
}
std::vector<data *> * data_handler::get_validation_data() {
	return validation_data;
}

int data_handler::get_class_count() {
	return num_classes;
}
```

common_data.h + common_data.cpp

```
#ifndef __COMMON_HPP
#define __COMMON_HPP
```

```
#pragma once

#include <vector>
#include "data.h"
#include "data_handler.h"

class common_data
{

	public:
	common_data();
	~common_data();
	void set_training_Data(std::vector<data *> * vect);
	void set_test_Data(std::vector<data *> * vect);
	void set_validation_Data(std::vector<data *> * vect);


	protected:
	std::vector <data *> * training_data;
	std::vector <data *> * test_data;
	std::vector <data *> * validation_data;

};


#endif // !__COMMON_HPP
```

```
#include "common_data.h"



common_data::common_data()
{
}


common_data::~common_data()
{
}
```

```
void common_data::set_training_Data(std::vector<data *> * vect) {
	training_data = vect;
}

void common_data::set_test_Data(std::vector<data *> * vect) {
	test_data = vect;
}

void common_data::set_validation_Data(std::vector<data *> * vect) {
	validation_data = vect;
}
```

Graphic.h + Graphic.cpp

```
#include "data.h";

#include <thread>

#include <SFML/Graphics.hpp>

#pragma once
class Graphic
{
public:
	Graphic();
	~Graphic();

	static void drawImages(data * query, data * input);
	static void drawImage(data * image, int * label);
};
```

```
#include "Graphic.h"



Graphic::Graphic()
{
}
```

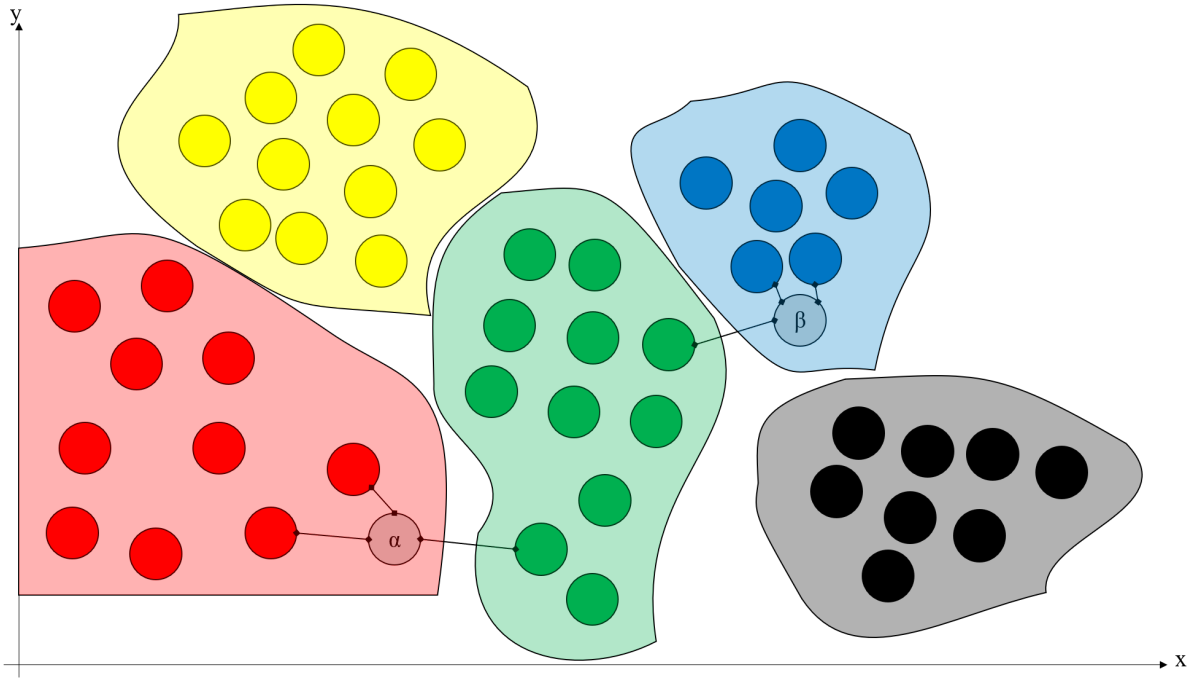
```
Graphic::~Graphic()
{
}

void Graphic::drawImages(data * query, data * input) {
	sf::RenderWindow gameWindow(sf::VideoMode(104, 84), "Images");
	gameWindow.setSize(sf::Vector2u(860, 760));

	sf::Image qImage;
	qImage.create(28, 28);
	for (int i = 0; i < query->get_feature_vector_size(); i++) {
		int c = query->get_feature_vector()->at(i);
		qImage.setPixel(i % 28, (i - (i % 28)) / 28, sf::Color(c, c, c, c));
	}
	qImage.saveToFile("qImage.png");
	sf::Texture qTexture;
	qTexture.loadFromImage(qImage);
	sf::Sprite qSprite(qTexture);
	qSprite.setPosition(100.0, 100.0);


	sf::Image iImage;
	iImage.create(28, 28);
	int x = 0;
	int y = 0;
	for (int i = 0; i < input->get_feature_vector_size(); i++) {
		int c = input->get_feature_vector()->at(i);
		iImage.setPixel(x, y, sf::Color(c, 0, 0, 255));
		x++;
		if (x == 28) { y++; x = 0; }
	}
	sf::Texture iTexture;
	iTexture.loadFromImage(iImage);
	sf::Sprite iSprite(iTexture);
	qSprite.setPosition(480.0, 500.0);

	while (gameWindow.isOpen()) {
		gameWindow.draw(iSprite);
```

```
                gameWindow.display();
        }

        printf("Loaded Images");
}

void Graphic::drawImage(data * image, int * label) {
        sf::RenderWindow gameWindow(sf::VideoMode(200, 200), "Images");
        gameWindow.setSize(sf::Vector2u(200, 200));

        sf::Image qImage;
        qImage.create(28, 28);
        for (int i = 0; i < image->get_feature_vector_size(); i++) {
                int c = image->get_feature_vector()->at(i);
                qImage.setPixel(i % 28, (i - (i % 28)) / 28, sf::Color(c, c, c, 255));
        }
        sf::Texture qTexture;
        qTexture.loadFromImage(qImage);
        sf::Sprite qSprite(qTexture);
        qSprite.setPosition(0, 0);
        qSprite.setScale(sf::Vector2f(5, 5));

        sf::Text tLabel;
        tLabel.setString("Prediction: " + std::string((char *)(label)));
        tLabel.setPosition(20, 150);
        tLabel.setFillColor(sf::Color::White);
        sf::Font f = sf::Font();
        f.loadFromFile("Fonts/arial.ttf");
        tLabel.setFont(f);

        gameWindow.draw(qSprite);
        gameWindow.draw(tLabel);


        gameWindow.display();
        std::this_thread::sleep_for(std::chrono::milliseconds(500));

}
```

KNN - K Nearest Neighbors

K Nearest Neighbors (KNN) is a simple and intuitive supervised machine learning algorithm. The concept of KNN is to place all the data, as points, in an ‘n’ dimensional void, and then compare new information to the ‘K’ closest data points (K Neighbors). The algorithm then classifies the new data according to the classes of the neighbors.

The following image shows many points on a two dimensional plane classified by their color. The translucent shapes show an approximation of the classification zones which the algorithm would come up with according to the given data. The diagram is of a “K = 3” KNN algorithm, meaning that there are three lines connecting points α and β to the three closest points. As visualized by the encompassing shapes, point α’s closest neighbors consist of two red points and one green point, meaning α is most likely a red point. Point β is very close to two blue points and one green point, and therefore is most likely a blue point as well.



Utility

KNN shines mostly in its simplicity of usage and implementation. While not being the best at classifying and not being too good for much else, contrary to a neural network, KNN starts working at a relatively good accuracy from the beginning. The one major setback of KNN is the necessity of having a substantial set of pre-classified examples for the classifier to compare with, though it does not require time to process and learn the data set, which is a great upside.

Final Result

The final result of the KNN classifier was acceptable, with an approximate 94% accuracy on the validation sample and 85% on the test sample when K was set to 8.

Code

```
#ifndef __KNN_H
#define __KNN_H
#pragma once

#include "common_data.h"

class knn : public common_data {

	int k;
	std::vector <data *> * neighbors;
	/*std::vector <data *> * training_data;
	std::vector <data *> * test_data;
	std::vector <data *> * validation_data;*/  // Now in common_data

	public:
	knn(int);
	knn();
	~knn();

	void find_knearest(data * query_point);
	/*void set_training_Data(std::vector<data *> * vect);
	void set_test_Data(std::vector<data *> * vect);
	void set_validation_Data(std::vector<data *> * vect);*/  // Now in common_data
	void set_k(int val);

	int predict();
	double calculate_distance(data * query_point, data * input);
	double validate_performance();
	double test_performance();
};



#endif // !__KNN_H
```

```
#include "knn.h"
#include <cmath>
#include <limits>
#include <map>
#include "stdint.h"
#include "data_handler.h"

#include "Graphic.h"

knn::knn(int val) {
	k = val;
}

knn::knn() {}
knn::~knn() {}

void knn::find_knearest(data * query_point) {
	neighbors = new std::vector<data *>;
	double min = std::numeric_limits<double>::max();
	double previous_min = min;
	int index = 0;

	for (int i = 0; i < k; i++) {
		if (i == 0) {
			for (int j = 0; j < training_data->size(); j++) {
				double distance = calculate_distance(query_point,
training_data->at(j));
				training_data->at(j)->set_distance(distance);
				if (distance < min) {
					min = distance;
					index = j;
				}
			}
			neighbors->push_back(training_data->at(index));
			previous_min = min;
			min = std::numeric_limits<double>::max();
		}
		else {
			for (int j = 0; j < training_data->size(); j++) {
				double distance = training_data->at(j)->get_distance(); //
calculate_distance(query_point, training_data->at(j));
```

```
                                if (distance > previous_min && distance < min) {
                                        min = distance;
                                        index = j;
                                }
                        }
                        neighbors->push_back(training_data->at(index));
                        previous_min = min;
                        min = std::numeric_limits<double>::max();
                }
        }
}
/*
void knn::set_training_Data(std::vector<data *> * vect) {
        training_data = vect;
}

void knn::set_test_Data(std::vector<data *> * vect) {
        test_data = vect;
}

void knn::set_validation_Data(std::vector<data *> * vect) {
        validation_data = vect;
}
*/  // Now in common_data
void knn::set_k(int val) {
        k = val;
}

int knn::predict() {
        std::map<uint8_t, int> class_freq;
        for (int i = 0; i < neighbors->size(); i++) { // Count the frequency of a class in the
neighboring points
                if (class_freq.find(neighbors->at(i)->get_label()) == class_freq.end()) {
                        class_freq[neighbors->at(i)->get_label()] = 1;
                } else {
                        class_freq[neighbors->at(i)->get_label()]++;
                }
        }

        int best = 0;
        int max = 0;
```

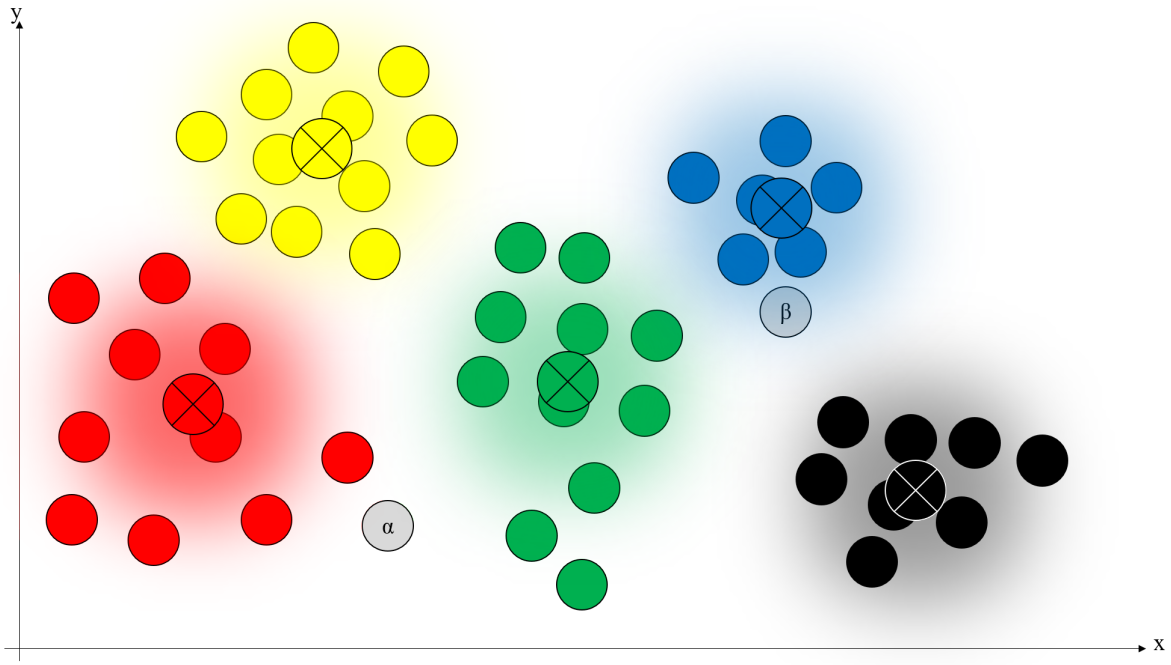
```
	for (std::pair<uint8_t, int> kv : class_freq) { // Find the most frequent class
		if (kv.second > max) {
			max = kv.second;
			best = kv.first;
		}
	}

	neighbors->clear();
	return best;
}

double knn::calculate_distance(data * query_point, data * input) {
	double distance = 0.0;
	if (query_point->get_feature_vector_size() != input->get_feature_vector_size()) {
		printf("Vector size mismatch");
		exit(1);
	}
//#ifdef EUCLID

	for (unsigned i = 0; i < query_point->get_feature_vector_size(); i++) {
		distance += pow(query_point->get_feature_vector()->at(i) -
input->get_feature_vector()->at(i),2);
	}
	distance = sqrt(distance);
	return distance;
}
double knn::validate_performance() {
	double current_performance = 0;
	int count = 0;
	int data_index = 0;

	for (data * query_point : * validation_data) {
		find_knearest(query_point);
		int prediction = predict();
		int asciiPred = prediction + 48;
		Graphic::drawImage(query_point, &asciiPred);
		printf("Guessed %d for %d\n", prediction, query_point->get_label());
		if (prediction == query_point->get_label()) {
			count++;
		}
```

```
                else {
                        //printf("Mistook %d for %d\n", query_point->get_label(),
prediction);
                }
                data_index++;

                printf("Current Performance: %.3f %%\n", ((double)count*100.0) /
((double)data_index));
        }

        current_performance = ((double)count*100.0) / ((double)validation_data->size());
        printf("Validation Performance for K = %d: %.3f %%\n", k, current_performance);
        return current_performance;
}

double knn::test_performance() {
        double current_performance = 0;
        int count = 0;
        //int data_index = 0;

        for (data * query_point : * test_data) {
                find_knearest(query_point);
                int prediction = predict();
                if (prediction == query_point->get_label()) {
                        count++;
                }
        }

        current_performance = ((double)count*100.0) / ((double)test_data->size());
        printf("Test Performance: %.3f %%\n", current_performance);
        return current_performance;
}

int main() {
        data_handler *dh = new data_handler();
        //dh->read_feature_vector("C:/Users/stein/Desktop/Research Project 2020/Neural
Networks/MNIST/Training Data/Training Data/train-images.idx3-ubyte");
        dh->read_feature_vector("../Training Data/train-images.idx3-ubyte");
        //dh->read_feature_labels("C:/Users/stein/Desktop/Research Project 2020/Neural
Networks/MNIST/Training Data/Training Data/train-labels.idx1-ubyte");
        dh->read_feature_labels("../Training Data/train-labels.idx1-ubyte");
```

```
	dh->split_data();
	dh->count_classes();

	knn * knearest = new knn();
	knearest->set_training_Data(dh->get_training_data());
	knearest->set_test_Data(dh->get_test_data());
	knearest->set_validation_Data(dh->get_validation_data());

	double performance = 0;
	double best_performance = 0;
	int best_k = 1;

	for (int k = 4; k <= 8; k++) {
		if (k == 1) {
			knearest->set_k(k);
			performance = knearest->validate_performance();
			best_performance = performance;
		}
		else {
			knearest->set_k(k);
			performance = knearest->validate_performance();
			if (performance > best_performance) {
				best_performance = performance;
				best_k = k;
			}
		}
	}

	knearest->set_k(best_k);
	knearest->test_performance();

	printf("Closing...");
}
```

K-Means Clustering

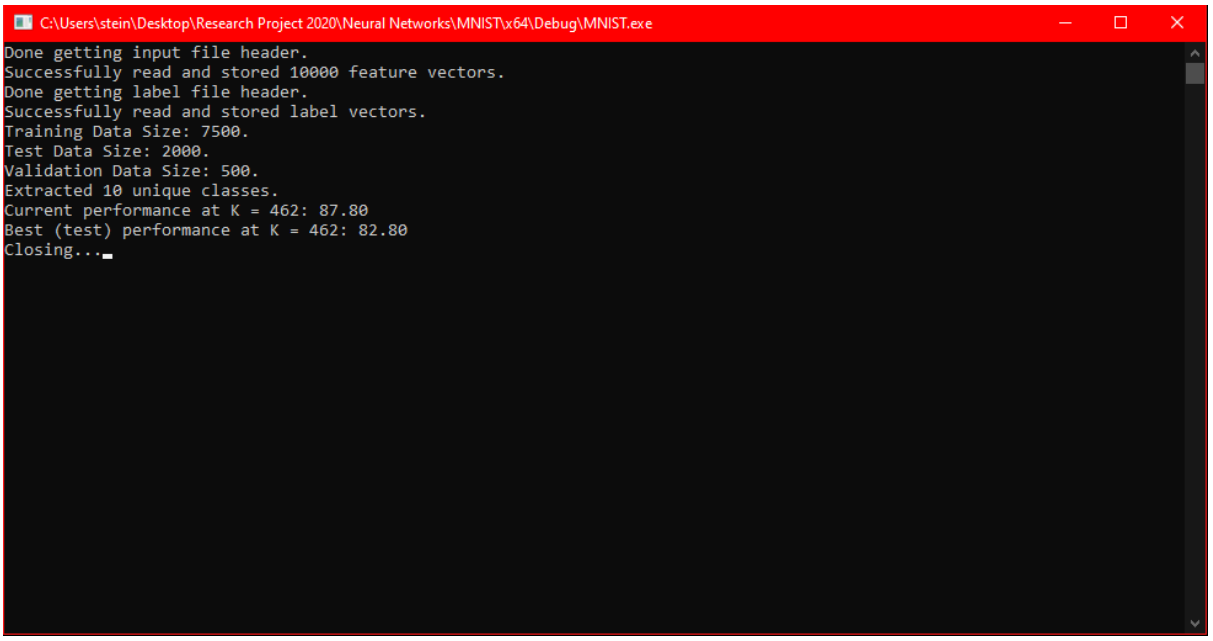
K-Means Clustering is relatively similar to KNN in the sense that it's also a supervised machine learning model. The main difference between the learning models is that K Nearest Neighbours tests each potential query point against all the points known to the algorithm and finds the K nearest ones, while K-Means tests query points against K centres of mass. K-Means initially loads all the known data points (the labeled training data) and splits it into K zones, each which corresponds to some label (for example, if we had two large clusters of points we could set K to 2 and assume that there is a high probability that another point close to one of the clusters would be of the same type as most of the points in the cluster).



Utility

K Means is significantly faster at classifying than KNN, which is a nifty upside for general testing. However, K Means was wildly inaccurate (usually below 80%) on all tests and validations I conducted. Therefore, K Means will not be developed further for use with audio classification.

Final Result



Clearly, K-Means is not ideal for this sort of classification, nor the rest of the project. However, it does have a clear advantage over KNN in terms of speed - once the clusters are constant, as K-Means only needs to compare a query point to the clusters, not every single other data point.

Code

```
#ifndef __KMEAN_HPP
#define __KMEAN_HPP

#pragma once

#include "common_data.h"
#include <unordered_set>
#include <limits>
#include <cstdlib>
#include <cmath>
#include <map>
#include <vector>

typedef struct cluster {

    std::vector<double> * centroid;
    std::vector<data *> * cluster_points;
    std::map<int, int> class_counts;
    int most_frequent_class;
```

```
    cluster(data * initial_point) {
        centroid = new std::vector<double>;
        cluster_points = new std::vector<data *>;
        for (auto value : *(initial_point->get_feature_vector())) {
            centroid->push_back(value);
        }
        cluster_points->push_back(initial_point);
        class_counts[initial_point->get_label()] = 1;
        most_frequent_class = initial_point->get_label();
    }

    void add_to_cluster(data * point) {
        int previous_size = cluster_points->size();
        cluster_points->push_back(point);
        for (int i = 0; i < centroid->size() - 1; i++) {
            double value = centroid->at(i);
            value *= previous_size;
            value += point->get_feature_vector()->at(i);
            value /= (double)cluster_points->size();
            centroid->at(i) = value;
        }

        if (class_counts.find(point->get_label()) == class_counts.end()) {
            class_counts[point->get_label()] = 1;
        } else {
            class_counts[point->get_label()]++;
        }
        set_most_frequent_class();
    }

    void set_most_frequent_class() {
        int best_class;
        int freq = 0;
        for (auto kv : class_counts) {
            if (kv.second > freq) {
                freq = kv.second;
                best_class = kv.first;
            }
        }
        most_frequent_class = best_class;
    }

} cluster_t;

class kmeans : public common_data {
    int num_clusters;
    std::vector<cluster_t *> * clusters;
    std::unordered_set<int> * used_indexes;
```

```
    public:
    kmeans(int k);
    kmeans();
    ~kmeans();
    void init_clusters();
    void init_clusters_for_each_class();
    void train();
    double euclidean_distance(std::vector<double> *, data *);
    double validate();
    double test();
};

#endif
```

```
#include "kmeans.h"

kmeans::kmeans() {}
kmeans::~kmeans() {}

kmeans::kmeans(int k) {
    num_clusters = k;
    clusters = new std::vector<cluster_t *>;
    used_indexes = new std::unordered_set<int>;
}
void kmeans::init_clusters() {
    for (int i = 0; i < num_clusters; i++) {
        int index = (rand() % training_data->size());
        while (used_indexes->find(index) != used_indexes->end()) {
        index = (rand() % training_data->size());
        }
        clusters->push_back(new cluster(training_data->at(index)));
        used_indexes->insert(index);
    }
}

void kmeans::init_clusters_for_each_class() {
    std::unordered_set<int> classes_used;
    for (int i = 0; i < training_data->size(); i++) {
        if (classes_used.find(training_data->at(i)->get_label()) == classes_used.end()) {
            clusters->push_back(new cluster_t(training_data->at(i)));
            classes_used.insert(training_data->at(i)->get_label());
            used_indexes->insert(i);
        }
    }
}
```

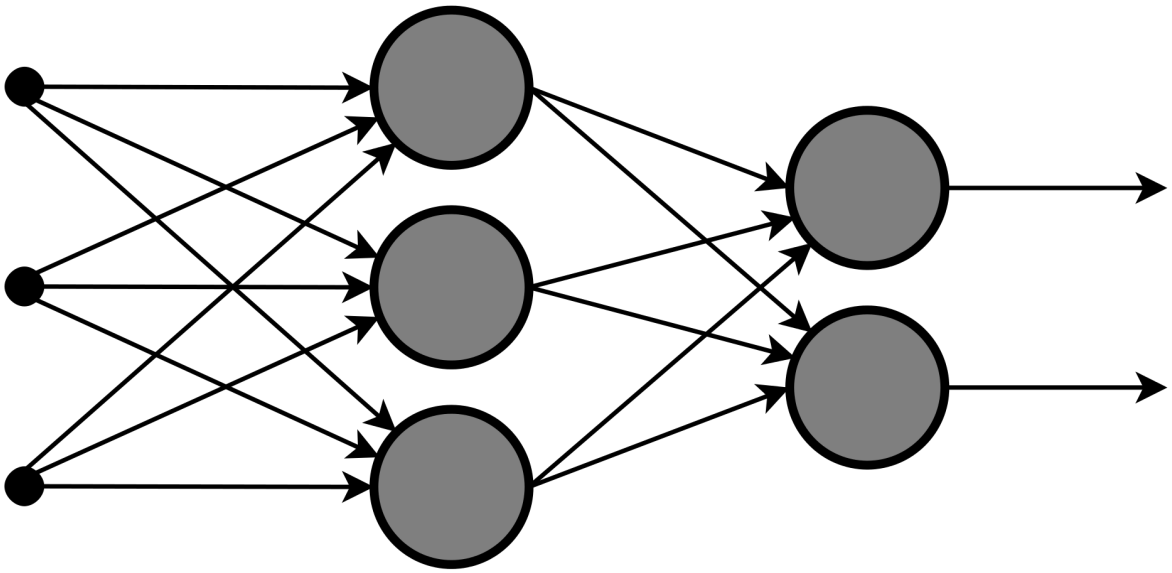
```
void kmeans::train() {
   while (used_indexes->size() < training_data->size()) {
      int index = (rand() % training_data->size());
      while (used_indexes->find(index) != used_indexes->end()) {
         index = (rand() % training_data->size());
      }

      double min_dist = std::numeric_limits<double>::max();
      int best_cluster = 0;
      for (int j = 0; j < clusters->size(); j++) {
         double current_dist = euclidean_distance(clusters->at(j)->centroid,
training_data->at(index));
         if (current_dist < min_dist) {
            min_dist = current_dist;
            best_cluster = j;
         }
      }
      clusters->at(best_cluster)->add_to_cluster(training_data->at(index));
      used_indexes->insert(index);
   }
}

double kmeans::euclidean_distance(std::vector<double> * centroid, data * point) {
   double dist = 0.0;
   for (int i = 0; i < centroid->size(); i++) {
      dist += pow(centroid->at(i) - point->get_feature_vector()->at(i), 2);
   }
   return sqrt(dist);
}

double kmeans::validate() {
   double num_correct = 0.0;
   for (auto query_point : *validation_data) {
      double min_dist = std::numeric_limits<double>::max();
      int best_cluster = 0;
      for (int j = 0; j < clusters->size(); j++) {
         double current_dist = euclidean_distance(clusters->at(j)->centroid, query_point);
         if (current_dist < min_dist) {
            min_dist = current_dist;
            best_cluster = j;
         }
      }
      if (clusters->at(best_cluster)->most_frequent_class == query_point->get_label()) {
         num_correct++;
      }
   }

   return 100.0 * (num_correct / (double)validation_data->size());
```

```
}
double kmeans::test() {
    double num_correct = 0.0;
    for (auto query_point : * test_data) {
        double min_dist = std::numeric_limits<double>::max();
        int best_cluster = 0;
        for (int j = 0; j < clusters->size(); j++) {
            double current_dist = euclidean_distance(clusters->at(j)->centroid, query_point);
            if (current_dist < min_dist) {
                min_dist = current_dist;
                best_cluster = j;
            }
        }
        if (clusters->at(best_cluster)->most_frequent_class == query_point->get_label()) {
            num_correct++;
        }
    }

    return 100.0 * (num_correct / (double)test_data->size());
}
```

Neural Network

Neural Networks are the classic example of Machine Learning. Neural Networks are the form of machine learning most similar to animals' organic learning, as the network of neurons is vaguely based on the neurons in our brains which are interconnected by synapses.

The idea is that each neuron is activated based on the weights and biases connected to it combined with the activation outputs of the neurons from the previous layer. The network evolves and learns by adjusting these weights and biases to output the desired values.



Conventionally, we group neurons into layers, which we can then say are connected to each other. Essentially, two connected layers means each neuron in the first layer is connected by a "synapses" with certain weights to each of the neurons in the following layer.

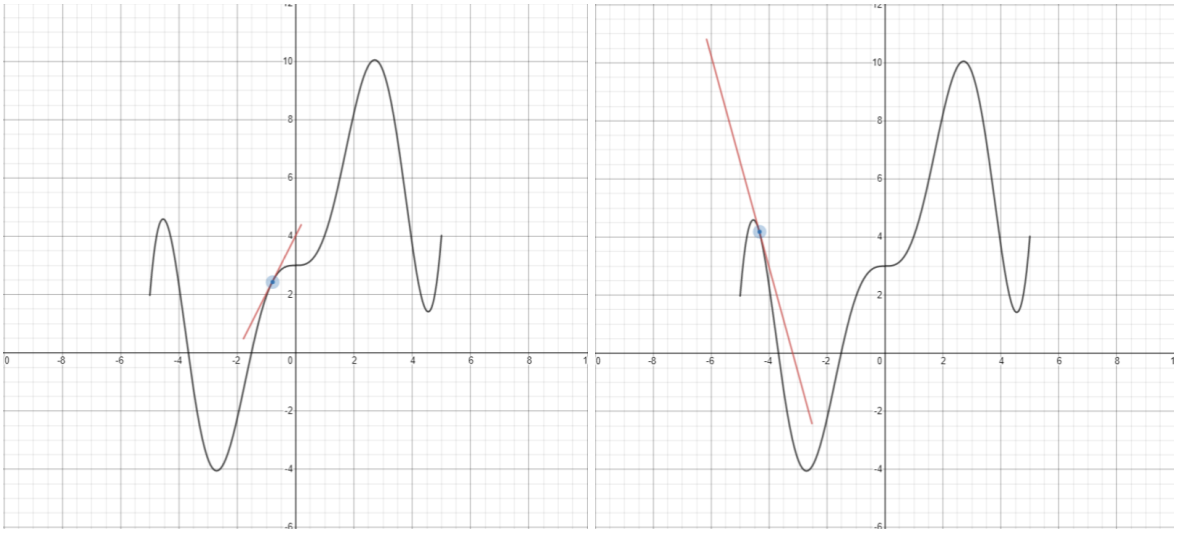
Training a neural network is where the complications escalate. What is implemented here is a backpropagation, which is an algorithm that computes the gradients of the loss function according to the difference between the outputs from the network and the desired outputs, and uses this gradient to find the minimum of the loss function - which would be the "smartest" network, the network who's weights and biases are as ideal as possible for the problem.

Gradient Descent

Gradient Descent is a way of computing which changes ("nudges") need to be made to the weights of a neuron in order to minimize the cost of the network as efficiently as possible. Gradient Descent in effect calculates the gradient vector which shows how sensitive the output of the network is to changes to each of the neurons in the current layer (doing so recursively is backpropagation).

We can visualize a two dimensional gradient descent, where the black line is our cost function, the blue dot is our weights and biases, and the red line is our gradient. What gradient descent tells us, is that if we follow the opposite of the gradient (-f'(x)) and move, as

in adjust the weights and biases, proportionally to its size (represented by a longer red line below), we will reach a local (ideally global) minimum of the cost function in a very efficient manner.



We can prove that this will converge onto a local minimum:

Let's call the cost function $F(x)$ and assume it is defined and differentiable for all values.

Let ∇F be the derivative of F.

Assume point a_n is our current location on the cost function, representing the weights and biases of the network.

Υ being our learning rate such that: $\Upsilon \in \mathbb{R}_+$ resolves to the next point a_{n+1} being:

$a_{n+1} = a_n - \Upsilon \nabla F(a_n)$

and therefore we have a monotonic sequence such that:

$F(a_1) \geq F(a_2) \geq F(a_3) \geq \ldots \geq F(a_n)$

meaning that$F(a_n)$ will be a local minimum of the cost function,

and as such, our weights and biases array - which is represented by a_n- will be semi-idealized to the problem

$\Longrightarrow$ The neural network has learnt to "solve" the given problem.

Backpropagation

Backpropagation is one of the core factors of machine learning. Backpropagation in essence is a method of changing the weights and biases of layers in a neural network depending on the layer after them. This method is crucial in training a network, since when training we cannot directly change the outputs of the last layer to fit our problem, rather we

change the weights and biases of all the neurons. Backpropagation allows us to quickly calculate how much to change the parameters of each neuron layer by layer simply by inputting our final output and our desired output.

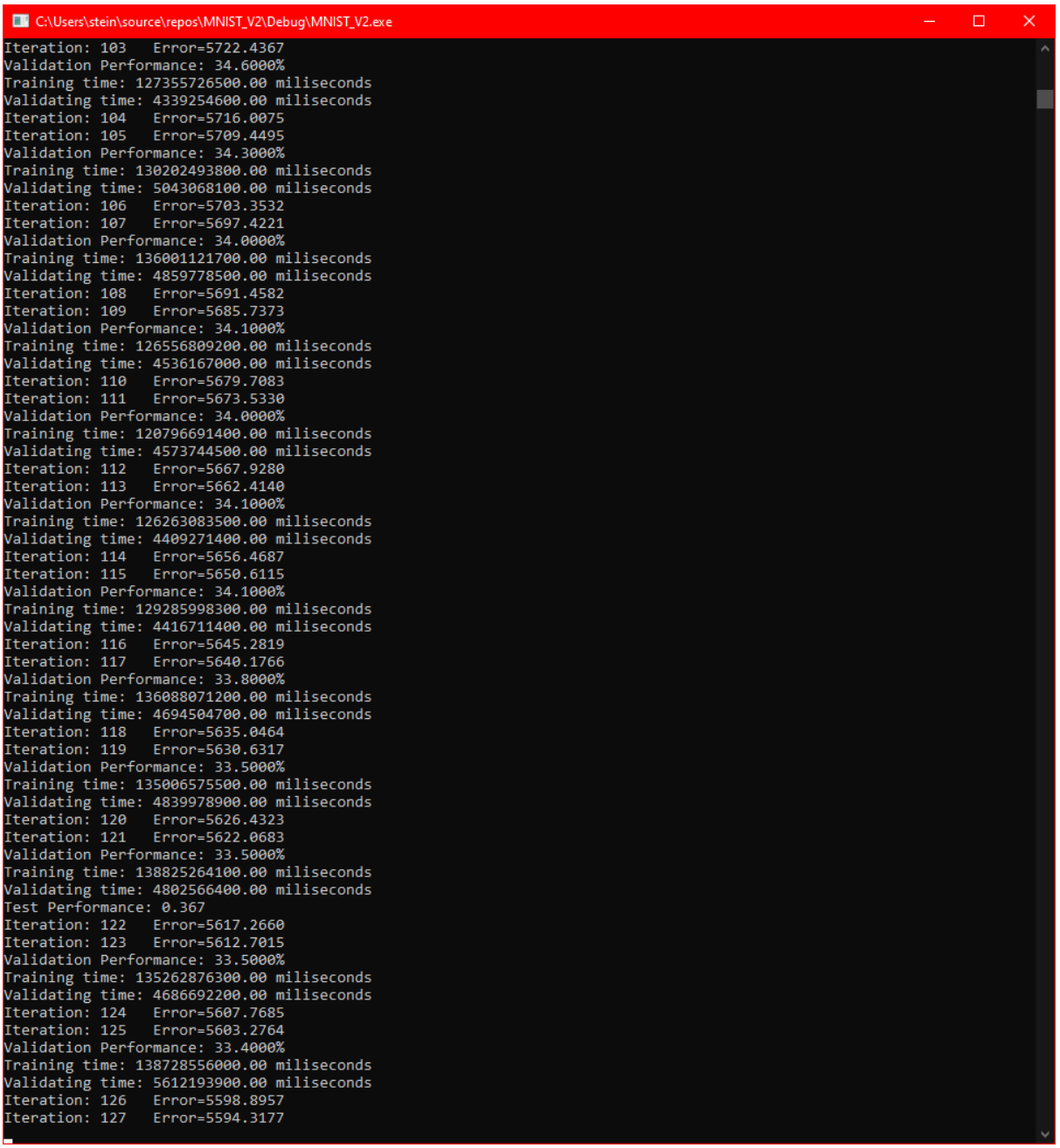
Backpropagation works by working backwards - from the last (output) layer, to the first layer - and calculating each time the effect that each neuron in that layer had on the output, and accordingly, should it be increased or decreased and by how much (this is done by calculating the error). Backpropagation then takes this calculation and uses it as the desired output for the previous layer, which in effect simplifies the network recursively so that we only ever update one layer as if it was the output layer and we knew exactly what outputs we wanted there.

Utility

The most alluring feature of neural networks is the fact that once they have been suited to the problem, they give predictions almost instantaneously. Contrary to KNN, which starts working with a solid accuracy immediately when launched, neural networks start off scoring poorly. Moreover, neural networks do not need an entire dataset to be loaded in memory in order to classify data; all they need is the weights and biases which best suit the problem - the weights and biases which produce the smallest cost function.

This means that once we adequately train a neural network, we can save the weights and biases it calculated and import them at a later date and immediately have a high accuracy, while maintaining a short prediction time.

Final Result



Here we can see a proof of concept for this neural network. Obviously the performance is not ideal. However, this is simply since we are using a very small network - only two hidden layers with ten neurons each. Nevertheless, we clearly see that the network is improving, the iteration error, which is the output of our error function, is going down, and the validation performance is going up.

The main limiting factor here is time; each training cycle takes approximately 12 minutes (disregard the time in the image, as a smaller training dataset was used to produce more results. Therefore, the actual error performance is also inaccurate). This is utterly

impractical, especially since for our final product we want a more complex network with many more calculations. See the faster version of this network here.

Code

```
#include "network.h"
#include "layer.h"
#include "DataHandler.h"
#include <numeric>
#include <algorithm>
#include <thread>


#include <chrono>
#include <utility>
typedef std::chrono::high_resolution_clock::time_point TimeVar;

#define duration(a) std::chrono::duration_cast<std::chrono::nanoseconds>(a).count()
#define timeNow() std::chrono::high_resolution_clock::now()

Network::Network(std::vector<int> spec, int inputSize, int numClasses, double
learningRate)
{
   for (int i = 0; i < spec.size(); i++)
   {
      if (i == 0)
         layers.push_back(new Layer(inputSize, spec.at(i)));
      else
         layers.push_back(new Layer(layers.at(i - 1)->neurons.size(), spec.at(i)));

   }
   layers.push_back(new Layer(layers.at(layers.size() - 1)->neurons.size(), numClasses));
   this->learningRate = learningRate;
}

Network::~Network() {}

double Network::activate(std::vector<double> weights, std::vector<double> input)
{
   double activation = weights.back(); // bias term
   for (int i = 0; i < weights.size() - 1; i++)
   {
      activation += weights[i] * input[i];
   }
   return activation;
}
```

```
double Network::transfer(double activation)
{
   return 1.0 / (1.0 + exp(-activation));
}

double Network::transferDerivative(std::vector<double> errors, std::vector<double>
inputs)
{
   double derivative = 0;

   for (int i = 0; i < errors.size(); i++) {
      derivative += errors[i] * inputs[i];
   }

   derivative *= 2.0 / errors.size();

   return derivative;
}

std::vector<double> Network::CalculateDerivative(Data * data, Layer * myLayer) {

   std::vector<double> errors1;
   std::vector<double> errors2;

   double gamma = 0.00000001;

   for (Neuron *n : myLayer->neurons) {
      double error = n->weights.at(n->weights.size() - 1); // bias
      for (int w = 0; w < n->weights.size() - 1; w++) {
         error += n->weights[w] * (data->getNormalizedFeatureVector()->at(w));
      }
      errors1.push_back(error);
   }

   for (Neuron *n : myLayer->neurons) {
      double error = n->weights.at(n->weights.size() - 1); // bias
      for (int w = 0; w < n->weights.size() - 1; w++) {
         error += n->weights[w] * (data->getNormalizedFeatureVector()->at(w) + gamma);
      }
      errors2.push_back(error);
   }

   std::vector<double> errors_difference;

   for (int i = 0; i < errors1.size(); i++) {
      errors_difference.push_back((errors2[i] - errors1[i]) / gamma);
   }
```

```
    return errors_difference;
}
std::vector<double> Network::CalculateDerivative(Layer * preLayer, Layer * myLayer) {

    std::vector<double> errors1;
    std::vector<double> errors2;

    double gamma = 0.00000001;

    for (Neuron *n : myLayer->neurons) {
        double error = n->weights.at(n->weights.size() - 1); // bias
        for (int w = 0; w < n->weights.size() - 1; w++) {
            error += n->weights[w] * preLayer->neurons[w]->output;
        }
        errors1.push_back(error);
    }

    for (Neuron *n : myLayer->neurons) {
        double error = n->weights.at(n->weights.size() - 1); // bias
        for (int w = 0; w < n->weights.size() - 1; w++) {
            error += n->weights[w] * (preLayer->neurons[w]->output + gamma);
        }
        errors2.push_back(error);
    }

    std::vector<double> errors_difference;

    for (int i = 0; i < errors1.size(); i++) {
        errors_difference.push_back((errors2[i] - errors1[i]) / gamma);
    }

    return errors_difference;
}

std::vector<double> Network::fprop(Data *data)
{
    std::vector<double> inputs = *data->getNormalizedFeatureVector();
    for (int i = 0; i < layers.size(); i++)
    {
        Layer *layer = layers.at(i);
        std::vector<double> newInputs;
        for (Neuron *n : layer->neurons)
        {
            double activation = this->activate(n->weights, inputs);
            n->output = this->transfer(activation);
            newInputs.push_back(n->output);
        }
        layer->layerOutputs = newInputs;
```

```
        inputs = newInputs;
    }
    return inputs; // output layer outputs
}

void Network::bprop(std::vector<double> deriv_errors) /// Backpropagation
{
    for (int i = layers.size() - 1; i >= 0; i--)
    {
        Layer *layer = layers.at(i);
        std::vector<double> errors;
        if (i != layers.size() - 1)
        {
            for (int j = 0; j < layer->neurons.size(); j++)
            {
                double error = 0.0;
                for (Neuron *n : layers.at(i + 1)->neurons)
                {
                    error += (n->weights.at(j) * n->delta);
                }
                errors.push_back(error);
            }
        }
        else {
            // If is last layer (i == layers.size()-1)
            errors = deriv_errors;
        }

        for (int j = 0; j < layer->neurons.size(); j++)
        {
            Neuron *n = layer->neurons.at(j);
            n->delta = errors[j];
        }
    }
}

void Network::updateWeights(Data *data)
{
    std::vector<double> inputs = *data->getNormalizedFeatureVector();
    for (int i = 0; i < layers.size(); i++)
    {
        if (i != 0)
        {
            for (Neuron *n : layers.at(i - 1)->neurons)
            {
                inputs.push_back(n->output);
            }
        }
```

```
        for (Neuron *n : layers.at(i)->neurons)
        {
            for (int j = 0; j < inputs.size(); j++)
            {
                n->weights.at(j) += this->learningRate * n->delta * inputs.at(j);
            }
            n->weights.back() += this->learningRate * n->delta;
        }
        inputs.clear();
    }
}

int Network::predict(Data * data)
{
    std::vector<double> outputs = fprop(data);
    return std::distance(outputs.begin(), std::max_element(outputs.begin(), outputs.end()));
}

void Network::train(int numEpochs, int iter)
{
    for (int i = 0; i < numEpochs; i++)
    {
        double sumError = 0.0;
        std::vector<double> deriv_error;

        int output_vector_size = this->trainingData->at(0)->getClassVector().size();
        for (int i = 0; i < output_vector_size; i++) { deriv_error.push_back(0.0); }

        for (Data *data : *this->trainingData)
        {
            std::vector<double> outputs = fprop(data);
            std::vector<int> expected = data->getClassVector();
            double tempErrorSum = 0.0;
            for (int j = 0; j < outputs.size(); j++)
            {
                tempErrorSum += pow((double)expected.at(j) - outputs.at(j), 2);
                deriv_error[j] += (double)expected.at(j) - outputs.at(j);
            }
            sumError += tempErrorSum;

            for (int j = 0; j < output_vector_size; j++)
            {
                deriv_error.at(j) /= this->trainingData->size();
                deriv_error.at(j) *= 2;
            }

            bprop(deriv_error);
            updateWeights(data);
```

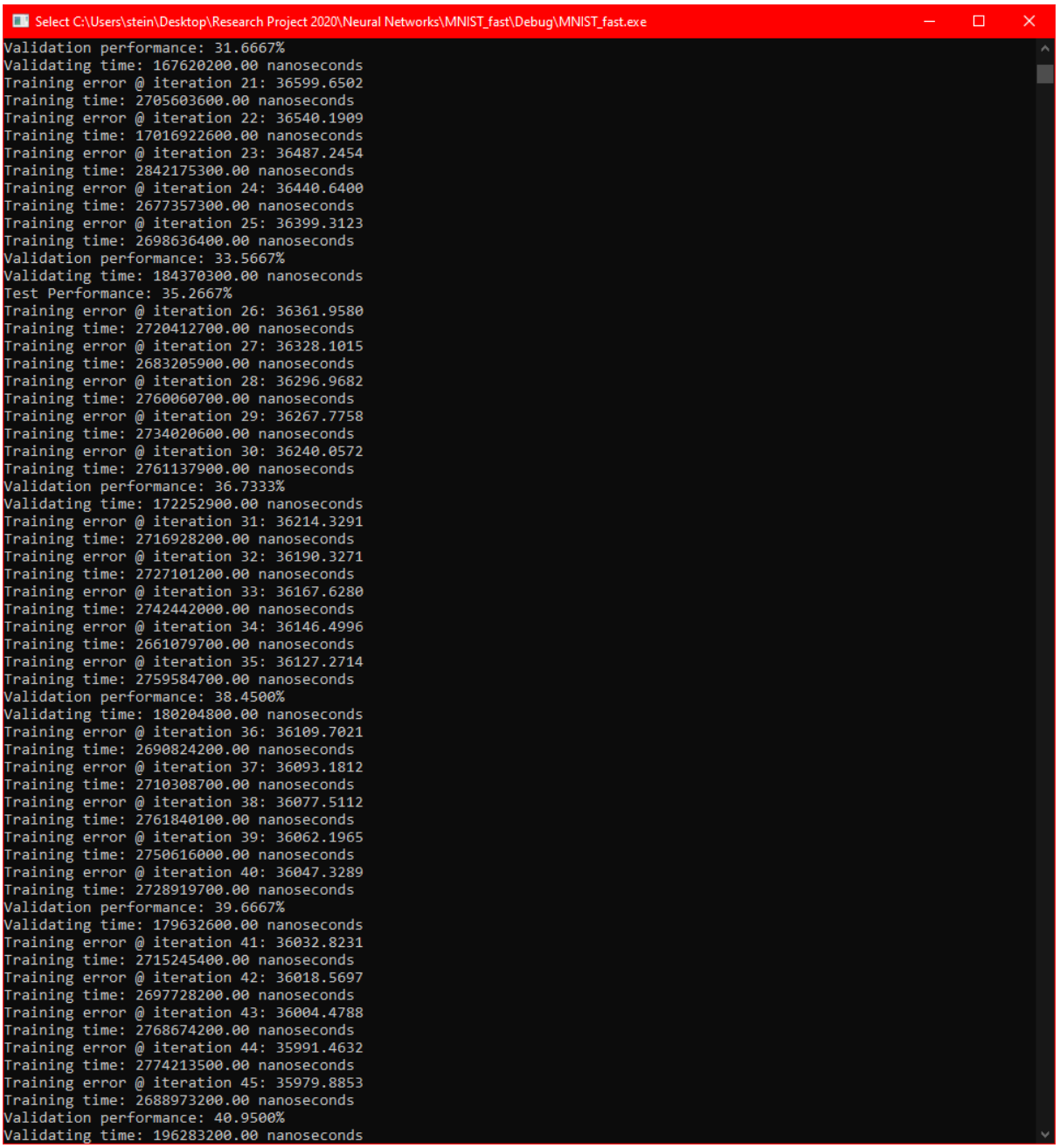
```
            for (int j = 0; j < output_vector_size; j++)
            {
                deriv_error.at(j) = 0;
            }
        }
        printf("Iteration: %d \t Error=%.4f\n", (iter * numEpochs) + i, sumError);
    }
}

double Network::test()
{
    double numCorrect = 0.0;
    double count = 0.0;
    for (Data *data : *this->testData)
    {
        count++;
        int index = predict(data);
        if (data->getClassVector().at(index) == 1) numCorrect++;
    }

    testPerformance = (numCorrect / count); // as decimal 0 -> 1
    return testPerformance;
}

void Network::validate()
{
    double numCorrect = 0.0;
    double count = 0.0;
    for (Data *data : *this->validationData)
    {
        count++;
        int index = predict(data);
        if (data->getClassVector().at(index) == 1) numCorrect++;
    }
    printf("Validation Performance: %.4f%%\n", (numCorrect*100) / count); // as
percentage 0.0000% -> 100.0000%
}

int main()
{
    DataHandler *dataHandler = new DataHandler();
    dataHandler->readInputData("C:\\Users\\stein\\Desktop\\Research Project 2020\\Neural
Networks\\MNIST\\Training Data\\train-images.idx3-ubyte");
    dataHandler->readLabelData("C:\\Users\\stein\\Desktop\\Research Project 2020\\Neural
Networks\\MNIST\\Training Data\\train-labels.idx1-ubyte");
    dataHandler->countClasses();
    dataHandler->splitData();
```

```
    std::vector<int> hiddenLayers = { 10,10 };
    Network *netw = new Network(
        hiddenLayers,
        dataHandler->getTrainingData()->at(0)->getNormalizedFeatureVector()->size(),
        dataHandler->getClassCounts(),
        0.25);
    netw->setTrainingData(dataHandler->getTrainingData());
    netw->setTestData(dataHandler->getTestData());
    netw->setValidationData(dataHandler->getValidationData());

    printf("Initial Validation: \n");
    netw->validate();
    printf("Now training...\n");

    for (int i = 0; i < 100; i++) {

        TimeVar t1 = timeNow();
        netw->train(2, i); // This takes 674923436000 nanoseconds = 674.92 seconds = 11:15
minutes
        double trainTime = duration(timeNow() - t1);

        TimeVar t2 = timeNow();
        netw->validate();
        double validateTime = duration(timeNow() - t2);

        printf("Training time: %.2f miliseconds\n", trainTime);
        printf("Validating time: %.2f miliseconds\n", validateTime);

        if (i % 10 == 0) {
            printf("Test Performance: %.3f\n", netw->test());
        }
    }
    printf("Test Performance: %.3f\n", netw->test());

    return 0;
}
```

Improved Neural Network

This version of the neural network is simply the previous version except with all the improvements listed in Evolvement - Practical Research. The main difference is that we used C code almost exclusively, instead of std::vector - which is a C++ class.

Final Result



As you can see, this version of the network reduced training time from ~674923436000 nanoseconds (674.92 seconds) to ~2661079700 nanoseconds (2.66 seconds); a 99.61% time reduction.

This improvement means we can compile and train a more complex network with more data samples in significantly less time.

Training the network for about an hour yielded a test performance of just over 80% with a simple network of one hidden layer with 32 neurons, which was plenty of proof to show that the network is in-fact *learning*. Thus, we conclude our proof-of-concept machine learning phase, and move to our actual goal.

Code

data.h + data.cpp

```
#pragma once

#include <vector>
#include <stdint.h>
#include <stdio.h>

class data
{
private:
    std::vector<uint8_t> * feature_vector;
    std::vector<double> * normalized_feature_vector;
    double * normalized_feature_array;
    uint32_t normalized_feature_array_size;
    std::vector<int> * class_vector;
    int * class_array;
    int class_array_size;
    uint8_t label;
    uint8_t enumerated_label;
    double distance;
public:
    void set_distance(double distance);
    void set_feature_vector(std::vector<uint8_t> * vect);
    void set_normalized_feature_vector(std::vector<double> * norm_vect);
    void update_normalized_feature_array();
    void set_class_vector(int counts);
    void set_class_array(int counts);
    void append_to_feature_vector(uint8_t val);
    void append_to_feature_vector(double val);
    void set_label(uint8_t label);
    void set_enumerated_label(uint8_t label);

    double get_distance();
    int get_feature_vector_size();
    int get_feature_array_size();
    uint8_t get_label();
    uint8_t get_enumerated_label();

    std::vector<uint8_t> * get_feature_vector();
    std::vector<double> * get_normalized_feature_vector();
    double * get_normalized_feature_array();
    std::vector<int> * get_class_vector();
    int * get_class_array();
```

```
    int get_class_array_size();

    void c_only();
};
```

```
#include "data.h"

void data::set_distance(double distance) {
    this->distance = distance;
}
void data::set_feature_vector(std::vector<uint8_t> * vect) {
    this->feature_vector = vect;
}
void data::set_normalized_feature_vector(std::vector<double> * norm_vect) {
    this->normalized_feature_vector = norm_vect;
}

void data::set_class_vector(int counts) {
    this->class_vector = new std::vector<int>();
    for (int i = 0; i < counts; i++)
    {
        if (i == label)
            this->class_vector->push_back(1);
        else
            this->class_vector->push_back(0);
    }
}
void data::set_class_array(int counts) {
    if (class_array != NULL) { free(class_array); }
    else { class_array = (int*)calloc(counts, sizeof(int)); }

    class_array[label] = 1;

    class_array_size = counts;
}

void data::append_to_feature_vector(uint8_t val) {
    feature_vector->push_back(val);
}
void data::append_to_feature_vector(double val) {
    normalized_feature_vector->push_back(val);
}
void data::set_label(uint8_t label) {
    this->label = label;
}
void data::set_enumerated_label(uint8_t label) {
    this->enumerated_label = label;
}
```

```
double data::get_distance() {
    return this->distance;
}

int data::get_feature_vector_size() {
    return feature_vector->size();
}
uint8_t data::get_label() {
    return this->label;
}
uint8_t data::get_enumerated_label() {
    return this->enumerated_label;
}

std::vector<uint8_t> * data::get_feature_vector() {
    return this->feature_vector;
}
std::vector<double> * data::get_normalized_feature_vector() {
    return this->normalized_feature_vector;
}
std::vector<int> * data::get_class_vector() {
    return this->class_vector;
}
int * data::get_class_array() {
    return this->class_array;
}

double * data::get_normalized_feature_array() {
    return normalized_feature_array;
}

void data::update_normalized_feature_array() {
    if (normalized_feature_array != NULL) { free(normalized_feature_array); }
    normalized_feature_array_size = normalized_feature_vector->size();
    normalized_feature_array = (double *)malloc(normalized_feature_array_size *
sizeof(double));

    for (int i = 0; i < normalized_feature_vector->size(); i++) {
        normalized_feature_array[i] = normalized_feature_vector->at(i);
    }
}

int data::get_feature_array_size() {
    return normalized_feature_array_size;
}

int data::get_class_array_size() {
```

```
    return class_array_size;
}

void data::c_only() {
    update_normalized_feature_array();

    if (class_array != NULL) { free(class_array); }
    class_array = (int *)malloc(class_vector->size() * sizeof(int));
    for (int i = 0; i < class_vector->size(); i++) {
        class_array[i] = class_vector->at(i);
    }

    delete normalized_feature_vector;
    delete feature_vector;

    delete class_vector;
}
```

data_handler.h + data_handler.cpp

```
#pragma once

#include "fstream"
#include "stdint.h"
#include "data.h"
#include <vector>
#include <string>
#include <map>
#include <unordered_set>
#include <math.h>
#include <algorithm>
#include <random>

class data_handler
{
private:
    std::vector<data *> * data_vector; // all the data vectors
    std::vector<data *> * training_data;
    std::vector<data *> * validation_data;
    std::vector<data *> * test_data;

    int class_counts;
    int feature_vector_size;
    std::map<uint8_t, int> class_from_int;
    std::map<std::string, int> class_from_string;
```

```
    uint32_t max_data_vector_size = 60000;

public:
    const double TRAIN_SET_PERCENT = 0.70;
    const double TEST_SET_PERCENT = 0.20;
    const double VALID_SET_PERCENT = 0.10;

    data_handler();     // ctor
    ~data_handler();    // dtor

    void read_input_data_file(std::string file_path);
    void read_input_label_file(std::string file_path);

    void split_data();
    void count_classes();
    void normalize_data();
    void log();

    int get_class_count();
    int get_data_vector_size();
    int get_training_data_size();
    int get_test_data_size();
    int get_validation_data_size();

    std::vector<data *> * get_training_data();
    std::vector<data *> * get_validation_data();
    std::vector<data *> * get_test_data();
    std::map<uint8_t, int> get_class_map();

    uint32_t format(const unsigned char* bytes);
};
```

```
#include "data_handler.h"

data_handler::data_handler() {
    data_vector = new std::vector<data *>;
    training_data = new std::vector<data *>;
    test_data = new std::vector<data *>;
    validation_data = new std::vector<data *>;
}
data_handler::~data_handler() {
}

void data_handler::read_input_data_file(std::string file_path) {
    uint32_t magic = 0;
    uint32_t num_images = 0;
    uint32_t num_rows = 0;
```

```
uint32_t num_cols = 0;

unsigned char bytes[4];
FILE *f;
errno_t err = fopen_s(&f, file_path.c_str(), "rb");
if (f)
{
   int i = 0;
   while (i < 4)
   {
      if (fread(bytes, sizeof(bytes), 1, f))
      {
         switch (i)
         {
         case 0:
            magic = format(bytes);
            i++;
            break;
         case 1:
            num_images = format(bytes);
            i++;
            break;
         case 2:
            num_rows = format(bytes);
            i++;
            break;
         case 3:
            num_cols = format(bytes);
            i++;
            break;
         }
      }
   }
   num_images = std::min(num_images, max_data_vector_size);
   printf("Done getting input data file header.\n");

   uint32_t image_size = num_rows * num_cols;
   for (i = 0; i < num_images; i++)
   {
      data * d = new data();
      d->set_feature_vector(new std::vector<uint8_t>());
      uint8_t element[1];
      for (int j = 0; j < image_size; j++)
      {
         if (fread(element, sizeof(element), 1, f))
         {
            d->append_to_feature_vector(element[0]);
         }
```

```
            }
            this->data_vector->push_back(d);
            d->set_class_vector(class_counts);
        }
        normalize_data();
        feature_vector_size = data_vector->at(0)->get_feature_vector_size();

        printf("Successfully read %lu data entries.\n", data_vector->size());
        printf("The Feature Vector Size is: %d\n", feature_vector_size);
    }
    else
    {
        printf("Could not open or read input vectors file!\n");
        printf("Now exiting...\n");
        exit(1);
    }
}
void data_handler::read_input_label_file(std::string file_path) {
    uint32_t magic = 0;
    uint32_t num_images = 0;
    unsigned char bytes[4];
    FILE *f;
    errno_t err = fopen_s(&f, file_path.c_str(), "rb");
    if (f)
    {
        int i = 0;
        while (i < 2)
        {
            if (fread(bytes, sizeof(bytes), 1, f))
            {
                switch (i)
                {
                case 0:
                    magic = format(bytes);
                    i++;
                    break;
                case 1:
                    num_images = format(bytes);
                    i++;
                    break;
                }
            }
        }

        printf("Done getting Label header.\n");

        num_images = std::min(num_images, max_data_vector_size);
```

```
        for (unsigned j = 0; j < num_images; j++)
        {
            uint8_t element[1];
            if (fread(element, sizeof(element), 1, f))
            {
                data_vector->at(j)->set_label(element[0]);
            }
        }

        printf("Successfully labeled %lu data entries.\n", num_images);
    }
    else
    {
        printf("Could not open or read input labels file!\n");
        printf("Now exiting...\n");
        exit(1);
    }
}

void data_handler::split_data() {
    std::unordered_set<int> used_indexes;
    int train_size = data_vector->size() * TRAIN_SET_PERCENT;
    int validation_size = data_vector->size() * VALID_SET_PERCENT;
    int test_size = data_vector->size() * TEST_SET_PERCENT;

    std::random_shuffle(data_vector->begin(), data_vector->end());

    // Training Data
    int count = 0;
    int index = 0;
    while (count < train_size)
    {
        training_data->push_back(data_vector->at(index++));
        count++;
    }

    // Validation Data
    count = 0;
    while (count < validation_size)
    {
        validation_data->push_back(data_vector->at(index++));
        count++;
    }

    // Test Data
    count = 0;
    while (count < test_size)
    {
```

```
        test_data->push_back(data_vector->at(index++));
        count++;
    }

    printf("Training Data Size: %lu.\n", training_data->size());
    printf("Validation Data Size: %lu.\n", validation_data->size());
    printf("Test Data Size: %lu.\n", test_data->size());

    if (training_data->size() + validation_data->size() + test_data->size() !=
data_vector->size()) {
        uint32_t amount_used = training_data->size() + validation_data->size() +
test_data->size();
        printf("Only %.2f%% of vectors were used. %d were not sorted.\n", 100.0 *
amount_used / data_vector->size(), data_vector->size() - amount_used);
    }
}

void data_handler::count_classes() {
    int count = 0;
    for (unsigned i = 0; i < data_vector->size(); i++)
    {
        if (class_from_int.find(data_vector->at(i)->get_label()) == class_from_int.end())
        {
            class_from_int[data_vector->at(i)->get_label()] = count;
            data_vector->at(i)->set_enumerated_label(count);
            count++;
        }
        else
        {

data_vector->at(i)->set_enumerated_label(class_from_int[data_vector->at(i)->get_label()])
;
        }
    }

    class_counts = count;
    for (data * data : *data_vector) {
        data->set_class_vector(class_counts);
        data->set_class_array(class_counts);
    }

    printf("Successfully Extracted %d Unique Classes.\n", class_counts);
}
void data_handler::normalize_data() {
    std::vector<double> mins, maxs;
    // fill min and max lists

    data * d = data_vector->at(0);
```

```
    for (uint8_t val : *(d->get_feature_vector()))
    {
        mins.push_back(val);
        maxs.push_back(val);
    }

    for (int i = 1; i < data_vector->size(); i++)
    {
        d = data_vector->at(i);
        for (int j = 0; j < d->get_feature_vector_size(); j++)
        {
            double value = (double)d->get_feature_vector()->at(j);
            if (value < mins.at(j)) mins[j] = value;
            if (value > maxs.at(j)) maxs[j] = value;
        }
    }

    // normalize data array
    for (int i = 0; i < data_vector->size(); i++)
    {
        data_vector->at(i)->set_normalized_feature_vector(new std::vector<double>());
        data_vector->at(i)->set_class_vector(class_counts);
        for (int j = 0; j < data_vector->at(i)->get_feature_vector_size(); j++)
        {
            // add normalized value to normalized_feature_vector
            if (maxs[j] - mins[j] == 0)
            { data_vector->at(i)->append_to_feature_vector(0.0); }
            else {

data_vector->at(i)->append_to_feature_vector((double)(data_vector->at(i)->get_feature_ve
ctor()->at(j) - mins[j]) / (maxs[j] - mins[j]));
            }
        }
    }
}
void data_handler::log() {

}

int data_handler::get_class_count() {
    return class_counts;
}
int data_handler::get_data_vector_size() {
    return data_vector->size();
}
int data_handler::get_training_data_size() { return training_data->size(); }
int data_handler::get_test_data_size() { return test_data->size(); }
int data_handler::get_validation_data_size() { return validation_data->size(); }
```

```
std::vector<data *> * data_handler::get_training_data() { return training_data; }
std::vector<data *> * data_handler::get_validation_data() { return validation_data; }
std::vector<data *> * data_handler::get_test_data() { return test_data; }
std::map<uint8_t, int> data_handler::get_class_map() { return class_from_int; }

uint32_t data_handler::format(const unsigned char* bytes)
{
    return (uint32_t)((bytes[0] << 24) |
        (bytes[1] << 16) |
        (bytes[2] << 8) |
        (bytes[3]));
}
```

common_data.h + common_data.cpp

```
#pragma once

#include "data.h"
#include <vector>

class common_data
{
protected:
    std::vector<data *> * training_data;
    std::vector<data *> * validation_data;
    std::vector<data *> * test_data;
public:
    void set_training_data(std::vector<data *> * vect);
    void set_validation_data(std::vector<data *> * vect);
    void set_test_data(std::vector<data *> * vect);

    std::vector<data *> * get_validation_data();
};
```

```
#include "common_data.h"

void common_data::set_training_data(std::vector<data *> * vect) {
    training_data = vect;
}
void common_data::set_validation_data(std::vector<data *> * vect) {
    validation_data = vect;
}
void common_data::set_test_data(std::vector<data *> * vect) {
```

```
    test_data = vect;
}

std::vector<data *> * common_data::get_validation_data() {
    return validation_data;
}
```

neuron.h + neuron.cpp

```
#pragma once

#include <stdio.h>
#include <vector>
#include <cmath>
#include <random>
#include <time.h>

class neuron
{
public:
    double output;
    double delta;
    std::vector<double> weights;
    double * weights_array;
    uint32_t weights_array_size;

    neuron(int, int); // ctor

    void initialize_weights(int); // randomize weight values

    double activate(std::vector<double> inputs);
    double activate(int inputs_size, double * inputs); // c

    void c_only();
};
```

```
#include "neuron.h"

double generateRandomNumber(double min, double max)
{
    double random = (double)rand() / RAND_MAX;
    return min + random * (max - min);
}

neuron::neuron(int prev_layer_size, int curr_layer_size)
```

```
{
    initialize_weights(prev_layer_size);
}
void neuron::initialize_weights(int prev_layer_size)
{
    for (int i = 0; i < prev_layer_size + 1; i++)
    {
        weights.push_back(generateRandomNumber(-1.0, 1.0));
    }
}

double neuron::activate(std::vector<double> inputs) {
    double activation = weights.back(); // bias
    for (int i = 0; i < weights.size() - 1; i++) {
        activation += weights[i] * inputs[i];
    }
    return activation;
}
double neuron::activate(int input_size, double * inputs) {
    double activation = weights_array[input_size]; // bias
    for (int i = 0; i < input_size; i++) {
        activation += weights_array[i] * inputs[i];
    }
    return activation;
}

void neuron::c_only() {
    if (weights.size() > 0) {
        weights_array = (double*)malloc(weights.size() * sizeof(double));
        weights_array_size = weights.size();
        for (int i = 0; i < weights.size(); i++) {
            weights_array[i] = weights.at(i);
        }
        weights.clear();
    }
}
```

layer.h + layer.cpp

```
#pragma once

#include "neuron.h"
#include <stdint.h>
#include <vector>
```

```
class layer
{
public:
   int current_layer_size;
   std::vector<neuron *> neurons;
   std::vector<double> layer_outputs;

   neuron ** neurons_array;
   int neurons_size;
   double * layer_outputs_array;

   layer(int prev_layer_size, int curr_layer_size); // ctor

   void c_only();
};
```

```
#include "layer.h"

layer::layer(int prev_layer_size, int curr_layer_size)
{
   for (int i = 0; i < curr_layer_size; i++)
   {
      neurons.push_back(new neuron(prev_layer_size, curr_layer_size));
   }

   this->current_layer_size = curr_layer_size;
}

void layer::c_only() {
   neurons_size = neurons.size();

   neurons_array = (neuron**)malloc(neurons_size * sizeof(neuron*));

   for (int i = 0; i < neurons_size; i++) {
      neurons_array[i] = neurons.at(i);
      neurons_array[i]->c_only();
   }

   layer_outputs_array = (double *)malloc(neurons_size * sizeof(double));

   neurons.clear();
   layer_outputs.clear();
}
```

network.h + network.cpp

```
#pragma once

#include "data.h"
#include "neuron.h"
#include "layer.h"
#include "common_data.h"
#include "data_handler.h"

#include <numeric>
#include <algorithm>
#include <thread>

class network : public common_data
{
public:
   std::vector<layer *> layers;
   layer ** layers_array;
   int layers_size;
   double learning_rate;
   double test_performance;

   network(std::vector<int> hidden_layers_specification, int, int, double); // ctor
   ~network(); // dtor

   void c_only();

   std::vector<double> fprop(data * d); // forward propagation
   void bprop(std::vector<double> deriv_errors); // back propagation
   void update_weights(data * data); // update weights after bprop

   int fprop_c(data * d, double ** output); // forward propagation - c | returns class count
   void bprop_c(double * deriv_errors); // back propagation - c
   void update_weights_c(data * data); // update weights after bprop - c

   double train(); // returns error
   double validate(); // return percentage correct
   double test(); // return percentage correct

   double train_c(); // returns error - c
   double validate_c(); // return percentage correct - c
   double test_c(); // return percentage correct - c

   double transfer_activation(double activat);

   int predict(data * data);
   int predict_c(data * data);

};
```

```
#include "network.h"


network::network(std::vector<int> hidden_layers_specification, int input_size, int
number_of_classes, double learning_rate) {
   for (int i = 0; i < hidden_layers_specification.size(); i++)
   {
      if (i == 0)
         layers.push_back(new layer(input_size, hidden_layers_specification.at(i)));
      else
         layers.push_back(new layer(layers.at(i - 1)->neurons.size(),
hidden_layers_specification.at(i)));

   }
   layers.push_back(new layer(layers.at(layers.size() - 1)->neurons.size(),
number_of_classes));

   this->learning_rate = learning_rate;
}
network::~network() {

}
std::vector<double> network::fprop(data * d) {
   std::vector<double> inputs = *d->get_normalized_feature_vector();

   for (int i = 0; i < layers.size(); i++)
   {
      layer * l = layers.at(i);
      std::vector<double> new_inputs;
      int neuron_index = 0;
      for (neuron * n : l->neurons)
      {
         double activation = n->activate(inputs);
         n->output = this->transfer_activation(activation);
         new_inputs.push_back(n->output);
      }
      l->layer_outputs = new_inputs;
      inputs = new_inputs;
   }
   return inputs; // output layer outputs
}
int network::fprop_c(data * d, double ** output) { // fprop with only c
   int inputs_size = d->get_feature_array_size();
   double * inputs = (double*)malloc(inputs_size * sizeof(double));

   memcpy(inputs, d->get_normalized_feature_array(), inputs_size * sizeof(double));

   for (int layer_index = 0; layer_index < layers_size; layer_index++) {
```

```
        layer * l = (layers_array[layer_index]);
        double * new_inputs = (double *)malloc(l->neurons_size * sizeof(double));
        for (int neuron_index = 0; neuron_index < l->neurons_size; neuron_index++) {
            neuron * n = (l->neurons_array[neuron_index]);
            double activation = n->activate(inputs_size, inputs);
            n->output = this->transfer_activation(activation);
            new_inputs[neuron_index] = n->output;
        }
        inputs_size = l->neurons_size;
        memcpy(l->layer_outputs_array, new_inputs, inputs_size * sizeof(double));
        inputs = (double *)realloc(inputs, inputs_size * sizeof(double));
        memcpy(inputs, new_inputs, inputs_size * sizeof(double));
        free(new_inputs);
    }
    *output = (double *)malloc(inputs_size * sizeof(double));
    memcpy(*output, inputs, inputs_size * sizeof(double));
    free(inputs);

    return inputs_size;
}

void network::bprop(std::vector<double> deriv_errors) {
    for (int i = layers.size() - 1; i >= 0; i--)
    {
        layer * l = layers.at(i);
        std::vector<double> errors;
        if (i != layers.size() - 1)
        {
            for (int j = 0; j < l->neurons.size(); j++)
            {
                double error = 0.0;
                for (neuron *n : layers.at(i + 1)->neurons)
                {
                    error += (n->weights.at(j) * n->delta);
                }
                errors.push_back(error);
            }
        }
        else {
            // If is last layer (i == layers.size()-1)
            errors = deriv_errors;
        }

        for (int j = 0; j < l->neurons.size(); j++)
        {
            neuron * n = l->neurons.at(j);
            n->delta = errors[j];
        }
```

```
    }
}
void network::bprop_c(double * deriv_errors) {
    double * errors;
    for (int i = layers_size - 1; i >= 0; i--)
    {
        layer * l = (layers_array[i]);

        if (i != layers_size - 1)
        {
            errors = (double *)malloc(l->neurons_size * sizeof(double));
            for (int j = 0; j < l->neurons_size; j++)
            {
                /// for each neuron in current layer.

                double error = 0.0;
                for (int k = 0; k < layers_array[i + 1]->neurons_size; k++)
                {
                    /// for each neuron in next layer
                    neuron * n = (layers_array[i + 1]->neurons_array[k]);
                    error += (n->weights_array[j] * n->delta);
                }
                errors[j] = error;
            }
        }
        else {
            // If is last layer (i == layers.size()-1)
            errors = deriv_errors;
        }

        for (int j = 0; j < l->neurons_size; j++)
        {
            neuron * n = (l->neurons_array[j]);
            n->delta = errors[j];
        }

        if (deriv_errors != errors) {
            free(errors);
        }
    }
}
void network::update_weights(data * d) {
    std::vector<double> inputs = * d->get_normalized_feature_vector();
    for (int i = 0; i < layers.size(); i++)
    {
        if (i != 0)
        {
            for (neuron *n : layers.at(i - 1)->neurons)
```

```
            {
                inputs.push_back(n->output);
            }
        }
        for (neuron *n : layers.at(i)->neurons)
        {
            for (int j = 0; j < inputs.size(); j++)
            {
                n->weights.at(j) += this->learning_rate * n->delta * inputs.at(j);
            }
            n->weights.back() += this->learning_rate * n->delta;
        }
        inputs.clear();
    }
}
void network::update_weights_c(data * d) {
    int inputs_size = d->get_feature_array_size();
    double * inputs = (double *)malloc(inputs_size * sizeof(double));
    memcpy(inputs, d->get_normalized_feature_array(), inputs_size * sizeof(double));

    for (int i = 0; i < layers_size; i++)
    {
        if (i != 0)
        {
            inputs_size = layers_array[i - 1]->neurons_size;
            inputs = (double *)realloc(inputs, inputs_size * sizeof(double));

            for (int j = 0; j < layers_array[i - 1]->neurons_size; j++)
            {
                neuron * n = (layers_array[i-1]->neurons_array[j]);
                inputs[i] = n->output;
            }
        }
        for (int j = 0; j < layers_array[i]->neurons_size; j++)
        {
            neuron * n = (layers_array[i]->neurons_array[j]);
            for (int k = 0; k < inputs_size; k++)
            {
                n->weights_array[k] += this->learning_rate * n->delta * inputs[k];
            }
            n->weights_array[inputs_size] += this->learning_rate * n->delta;
        }
    }

    free(inputs);
}

double network::train() {
```

```
    double sum_error = 0.0;
    std::vector<double> deriv_error;

    int output_vector_size = this->training_data->at(0)->get_class_vector()->size();
    for (int i = 0; i < output_vector_size; i++) { deriv_error.push_back(0.0); }

    for (data * d : *this->training_data)
    {
        std::vector<double> outputs = fprop(d);
        std::vector<int> expected = *(d->get_class_vector());
        double tmp_sum_error = 0.0;
        for (int j = 0; j < outputs.size(); j++)
        {
            tmp_sum_error += pow((double)expected.at(j) - outputs.at(j), 2);
            deriv_error[j] += (double)expected.at(j) - outputs.at(j);
        }
        sum_error += tmp_sum_error;

        for (int j = 0; j < output_vector_size; j++)
        {
            deriv_error.at(j) /= this->training_data->size();
            deriv_error.at(j) *= 2;
        }

        bprop(deriv_error);
        update_weights(d);

        for (int j = 0; j < output_vector_size; j++)
        {
            deriv_error.at(j) = 0;
        }
    }

    return sum_error;
}
double network::validate() {
    double num_correct = 0.0;
    double count = 0.0;
    for (data * d : *this->validation_data)
    {
        count++;
        int index = predict(d);
        if (d->get_class_vector()->at(index) == 1) { num_correct++; }
    }

    return num_correct / count;
}
double network::test() {
```

```
    double num_correct = 0.0;
    double count = 0.0;
    for (data * d : *this->test_data)
    {
        count++;
        int index = predict(d);
        if (d->get_class_vector()->at(index) == 1) { num_correct++; }
    }

    return num_correct / count;
}

double network::train_c() {
    double sum_error = 0.0;
    int output_vector_size = this->training_data->at(0)->get_class_array_size();
    double * deriv_error = (double *)calloc(output_vector_size, sizeof(double));

    for (data * d : *this->training_data)
    {
        double * outputs;
        fprop_c(d, &outputs);

        int * expected = d->get_class_array();
        double tmp_sum_error = 0.0;
        for (int j = 0; j < output_vector_size; j++)
        {
            double exp = (double)expected[j];
            double out = outputs[j];
            tmp_sum_error += pow(exp - out, 2);
            deriv_error[j] += exp - out;
        }
        sum_error += tmp_sum_error;

        for (int j = 0; j < output_vector_size; j++)
        {
            deriv_error[j] /= this->training_data->size();
            deriv_error[j] *= 2;
        }

        bprop_c(deriv_error);
        update_weights_c(d);

        for (int j = 0; j < output_vector_size; j++)
        {
            deriv_error[j] = 0;
        }

        free(outputs);
```

```
    }

    free(deriv_error);

    return sum_error;
}
double network::validate_c() {
    double num_correct = 0.0;
    double count = 0.0;
    for (data * d : *this->validation_data)
    {
        count++;
        int index = predict_c(d);
        if (d->get_class_array()[index] == 1) { num_correct++; }
    }

    return num_correct / count;
}
double network::test_c() {
    double num_correct = 0.0;
    double count = 0.0;
    for (data * d : *this->test_data)
    {
        count++;
        int index = predict_c(d);
        if (d->get_class_array()[index] == 1) { num_correct++; }
    }

    return num_correct / count;
}

double network::transfer_activation(double activat) {
    return 1.0 / (1.0 + exp(-activat));
}

int network::predict(data * data) {
    std::vector<double> outputs = fprop(data);
    return std::distance(outputs.begin(), std::max_element(outputs.begin(), outputs.end()));
}
int network::predict_c(data * data) {
    double * outputs;
    int class_count = fprop_c(data, &outputs);

    int max_ind = 0;
    double max = 0;
    for (int i = class_count - 1; i >= 0; i--) {
        if (outputs[i] > max) { max_ind = i; max = outputs[i]; }
    }
```

```
    free(outputs);

    return max_ind;
}

void network::c_only() {
    layers_size = layers.size();
    layers_array = (layer **)malloc(layers_size * sizeof(layer *));
    for (int i = 0; i < layers_size; i++) {
        layers_array[i] = (layers.at(i));
    }

    layers.clear();

    for (int i = 0; i < training_data->size(); i++) {
        training_data->at(i)->c_only();
    }
    for (int i = 0; i < validation_data->size(); i++) {
        validation_data->at(i)->c_only();
    }
    for (int i = 0; i < test_data->size(); i++) {
        test_data->at(i)->c_only();
    }

    for (int i = 0; i < layers_size; i++) {
        layer * l = layers_array[i];
        l->c_only();
    }

    printf("Network is now C only!\n");
}
```

main.cpp

```
#include <iostream>

#include "network.h"

#include <chrono>
#include <utility>
typedef std::chrono::high_resolution_clock::time_point TimeVar;

#define duration(a) std::chrono::duration_cast<std::chrono::nanoseconds>(a).count()
#define timeNow() std::chrono::high_resolution_clock::now()
```

```
int main()
{
   srand(time(NULL));

   data_handler * _data_handler = new data_handler();
   _data_handler->read_input_data_file("C:\\Users\\stein\\Desktop\\Research Project
2020\\Neural Networks\\MNIST\\Training Data\\train-images.idx3-ubyte");
   _data_handler->read_input_label_file("C:\\Users\\stein\\Desktop\\Research Project
2020\\Neural Networks\\MNIST\\Training Data\\train-labels.idx1-ubyte");
   _data_handler->count_classes();
   _data_handler->split_data();

   std::vector<int> hidden_layers = { 10,10 };
   network * netw = new network(
      hidden_layers,
      _data_handler->get_training_data()->at(0)->get_feature_vector_size(),
      _data_handler->get_class_count(),
      0.25);

   netw->set_training_data(_data_handler->get_training_data());
   netw->set_validation_data(_data_handler->get_validation_data());
   netw->set_test_data(_data_handler->get_test_data());


   printf("Initial Validation: \n");
   printf("Validation performance: %.4f%%\n", 100.0 * netw->validate());

   netw->c_only();

   printf("Initial Validation - C: \n");
   printf("Validation performance: %.4f%%\n", 100.0 * netw->validate_c());
   printf("Now training...\n");



   for (int i = 0; i < 100000; i++) {

      TimeVar t1 = timeNow();
      printf("Training error @ iteration %d: %.4f\n", i, netw->train_c());
      double train_time = duration(timeNow() - t1);

      printf("Training time: %.2f nanoseconds\n", train_time);

      if (i % 5 == 0) {
         TimeVar t2 = timeNow();
         printf("Validation performance: %.4f%%\n", 100.0 * netw->validate_c());
         double validate_time = duration(timeNow() - t2);
```

```
            printf("Validating time: %.2f nanoseconds\n", validate_time);
        }


        if (i % 25 == 0) {
            printf("Test Performance: %.4f%%\n", 100.0 * netw->test_c());
        }
    }
    printf("Test Performance: %.4f%%\n", 100.0 * netw->test_c());

    return 0;
}
```

FFT - Fast Fourier Transform

Fast Fourier Transform is a fast algorithm to calculate the Discrete Fourier Transform of a complex set. FFT improves DFT, as it reduces the time complexity from $O(N^2)$ to $O(N \log(N))$.

DFT - Discrete Fourier Transform

DFT transforms a finite sequence of equally-spaced samples of a function into a same-length sequence of equally spaced samples of the discrete-time Fourier Transform. DFT can be used to calculate the coefficients of the complex sinusoids which built the input.

DFT can be regarded as transforming a functions' time/space domain into a frequency domain.

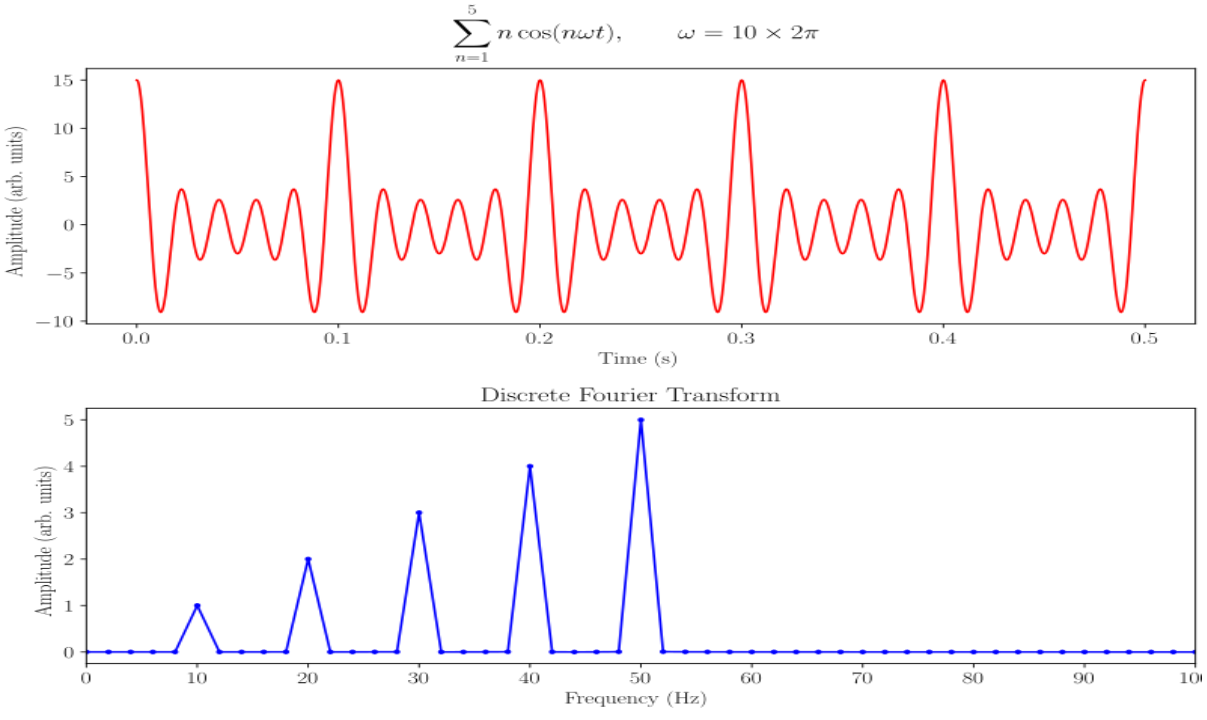
The Discrete Fourier Transform of $\{x_n\} := x_0, x_1, \ldots, x_{N-1}$ to $\{X_k\} := X_0, X_1, \ldots, X_{N-1}$ is defined by:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{(-\frac{i2\pi}{N} kn)}$$

and following Euler's formula:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot \left[cos(\frac{2\pi}{N} kn) - i \cdot sin(\frac{2\pi}{N} kn) \right]$$

We will be using FFT to transform audio data into a fixed size array of the relative amplitudes on each frequency from 1 to 4000. The input we will be giving the FFT function is the time-amplitude coordinates of an audio wave, and it will return an array with the amplitudes of the frequencies. A simple example of a fourier transform:



We will be using the classic Cooley-Tukey FFT algorithm. The Cooley-Tukey algorithm recursively breaks down a Discrete Fourier Transform into Discrete Fourier Transforms of smaller sizes, a divide-and-conquer approach. The divide-and-conquer algorithmic approach is a design paradigm which recursively breaks down a problem into multiple subproblems of a similar type until these subproblems are elementary enough to be directly solved. The solutions to the subproblems are then combined to solve the original problem.

The method used here is to divide the Discrete Transform into two sets of size N/2 (N being the size of the input set at each recursive calling) each time. Each calling then combines, by summation, the results into the passed array.

The code we will use to perform this Cooley-Tukey Fast Fourier Transform is:

```
int fft(CArray& x) {
   const size_t N = x.size();
   if (N <= 1) return 1;

   // divide
   CArray even = x[std::slice(0, N/2, 2)];
   CArray  odd = x[std::slice(1, N/2, 2)];

   // conquer
   fft(even);
   fft(odd);

   // combine
   for (size_t k = 0; k < N/2; ++k)
   {
      Complex t = std::polar(1.0, -2 * PI * k / N) * odd[k];
      x[k    ] = even[k] + t;
      x[k+N/2] = even[k] - t;
   }

   return 0;
}
```

And in order to alleviate the tension on the rest of the code, we built a wrapper class for the Fast Fourier Transform which will take care of all the processing.

Code - FFT.h + FFT.cpp:

FFT.h

```
#ifndef FFT_H
```

```
#define FFT_H

#include <complex>
#include <iostream>
#include <valarray>
#include <float.h>
#include <vector>
#include <algorithm>

typedef std::complex<double> Complex;
typedef std::valarray<Complex> CArray;

class FFT
{
   public:
      FFT();
      virtual ~FFT();

      int LoadWave(double[]);
      int AppendToWave(double);

      std::vector<Complex> * FourierTransfer(uint32_t samplingRate); /// Returns the
frequency outputs for an entire
      std::vector<Complex> * FourierTransfer_Part(uint32_t samplintRate, uint32_t index);

      const double PI = 3.14159265358979323846;

      Complex * getOriginalWave();
      int getOriginalWaveSize();

      Complex * getFrequencyOutput();
      std::vector<double> * ActualValues;
      std::vector<Complex> * FrequencyOutput;

      std::vector<std::vector<double> * > * splitValues;

      std::vector<double> * getNormalizedFrequency(); // Frequencies ranged from 0 to
255. Relative to Frequency output, obviously.
      std::vector<std::vector<double> * > * OrderFrequencyOutputs(std::vector<Complex>
* freqOut);

      double peakHigh;
      double peakLow;
      double range;

      std::vector<std::vector<double>> * peaks;
      std::vector<std::vector<double>> * NormalizedPeaks;
```

```
        int getK() { return k; }

        uint32_t MaxFrequency(); // The highest frequency to check.

    protected:

    private:
        int fft(CArray& x);
        int inverse_fft(CArray& x);

        int SplitValues();

        int ReloadArrays();

        int k = 25; // Amount of peaks to keep track of

        uint32_t FrequencyRange = 4096; // The highest frequency to check. (To speed up
processing)
        uint32_t SamplingRate = 8000;

        void ValidatePeak(double ind, double p);
        void ValidateNormalizedPeak(double ind, double p);
        void ResetPeaks();
        void ResetNormalizedPeaks();



};

#endif // FFT_H
```

FFT.cpp

```
#include "FFT.h"

FFT::FFT()  {
    //ctor
    ActualValues = new std::vector<double>();
}


FFT::~FFT() {
    //dtor
}
```

```
void FFT::ResetPeaks() {
   peaks = new std::vector<std::vector<double>>();
   std::vector<double> demoPeak = std::vector<double>(); demoPeak.push_back(-1.0);
demoPeak.push_back(-DBL_MAX);
   for (int i = 0; i < k; i++) { peaks->push_back(demoPeak); }
}

void FFT::ResetNormalizedPeaks() {
   NormalizedPeaks = new std::vector<std::vector<double>>();
   std::vector<double> demoPeak2 = std::vector<double>(); demoPeak2.push_back(-1.0);
demoPeak2.push_back(-DBL_MAX);
   for (int i = 0; i < k; i++) { NormalizedPeaks->push_back(demoPeak2); }
}

int FFT::fft(CArray& x) {
   const size_t N = x.size();
   if (N <= 1) return 1;

   // divide
   CArray even = x[std::slice(0, N/2, 2)];
   CArray  odd = x[std::slice(1, N/2, 2)];

   // conquer
   fft(even);
   fft(odd);

   // combine
   for (size_t k = 0; k < N/2; ++k)
   {
      Complex t = std::polar(1.0, -2 * PI * k / N) * odd[k];
      x[k    ] = even[k] + t;
      x[k+N/2] = even[k] - t;
   }

   return 0;
}

int FFT::inverse_fft(CArray& x) {
   // conjugate the complex numbers
   x = x.apply(std::conj);

   // forward fft
   fft( x );

   // conjugate the complex numbers again
   x = x.apply(std::conj);
```

```
    // scale the numbers
    x /= x.size();

    return 0;
}

int FFT::SplitValues() {
    splitValues = new std::vector<std::vector<double> *>();

    int amtSplits = std::ceil(ActualValues->size() / SamplingRate);

    for (int i = 0; i < amtSplits; i++) {
        std::vector<double> * partSamps = new std::vector<double>();

        for (int x = 0; x < SamplingRate; x++) {
            partSamps->push_back(ActualValues->at((i * SamplingRate) + x));
        }

        splitValues->push_back(partSamps);
    }

    return 0;
}

int FFT::LoadWave(double wave[]) {
    for (int i = 0; i < sizeof(wave) / sizeof(wave[0]); i++) {
        ActualValues->push_back(wave[i]);
    }

    return (ActualValues->size() == sizeof(wave) / sizeof(wave[0]));
}

int FFT::AppendToWave(double val) {
    ActualValues->push_back(val);
    return 0;
}

std::vector<Complex> * FFT::FourierTransfer(uint32_t samplingRate) {
    SamplingRate = samplingRate;
    SplitValues();

    FrequencyOutput = new std::vector<Complex>();

    for (int i = 0; i < samplingRate; i++) { FrequencyOutput->push_back(0); }

    int amtSplits = std::ceil(ActualValues->size() / SamplingRate);

    for (int splitInd = 0; splitInd < amtSplits; splitInd++) {
```

```
    std::vector<double> * liveValues = splitValues->at(splitInd);

    Complex originalWave[8000];
    for (int i = 0; i < liveValues->size(); i++) {
      originalWave[i] = liveValues->at(i);
    }

    CArray data(originalWave, sizeof(originalWave) / sizeof(originalWave[0]));
    FFT::fft(data);

    std::vector<Complex> * liveFrequencyOutput = new std::vector<Complex>();

    double pHigh = data[0].real();
    double pLow = data[0].real();

    int pHighInd = 0;
    int pLowInd = 0;

    ResetNormalizedPeaks();

    for (int i = 0; i < MaxFrequency(); i++){
      pHigh = std::max(pHigh, data[i].real());
      if (pHigh == data[i].real()) { pHighInd = i; }
      pLow = std::min(pLow, data[i].real());
      if (pLow == data[i].real()) { pLowInd = i; }
      liveFrequencyOutput->push_back(data[i]);
      ValidatePeak((double)(i), data[i].real());
      FrequencyOutput->at(i) = FrequencyOutput->at(i) + data[i].real();
    }

    peakHigh = pHigh;
    peakLow = pLow;
    range = pHigh - pLow;
    std::cout << "Sample #" << splitInd << ": {" << pHighInd << ", " << peakHigh << "},
{" << pLowInd << ", " << peakLow << "}" << std::endl;
  }

  for (int i = 0; i < SamplingRate; i++) { FrequencyOutput->at(i) =
(double)FrequencyOutput->at(i).real() / (double)amtSplits; }

  return FrequencyOutput;
}

std::vector<Complex> * FFT::FourierTransfer_Part(uint32_t samplingRate, uint32_t
index) {
  SamplingRate = samplingRate;

  Complex originalWave[8000];
```

```
    for (int i = 0; i < SamplingRate; i++) {
        originalWave[i] = ActualValues->at((index * SamplingRate) + i);
    }

    CArray data(originalWave, sizeof(originalWave) / sizeof(originalWave[0]));
    FFT::fft(data);

    std::vector<Complex> * liveFrequencyOutput = new std::vector<Complex>();

    double pHigh = std::abs(data[0].real());
    double pLow = std::abs(data[0].real());

    int pHighInd = 0;
    int pLowInd = 0;

    for (int i = 0; i < MaxFrequency(); i++) {
        data[i] = std::abs(data[i].real());

        pHigh = std::max(pHigh, data[i].real());
        if (pHigh == data[i].real()) { pHighInd = i; }
        pLow = std::min(pLow, data[i].real());
        if (pLow == data[i].real()) { pLowInd = i; }
    }
    for (int i = 0; i < MaxFrequency(); i++) {
        liveFrequencyOutput->push_back(((255.0 * (data[i] - pLow)) / (pHigh - pLow)));    //
Normalize the frequency amplitudes (convert arbitrary values to 0-255)
    }

    std::cout << "Sample #" << index << ": {" << pHighInd << ", " << pHigh << "}, {" <<
pLowInd << ", " << pLow << "}" << std::endl;

    return liveFrequencyOutput;
}

std::vector<std::vector<double> * > *
FFT::OrderFrequencyOutputs(std::vector<Complex> * freqOut) {
    std::vector<std::vector<double> * > * ordFreqOut = new
std::vector<std::vector<double> * >();

    std::vector<double> * tmpV = new std::vector<double>(); tmpV->push_back(0);
tmpV->push_back(-1);
    for ( int i = 0; i < k; i++) { ordFreqOut->push_back(tmpV); }

    //uint32_t usedIndexs[freqOut->size()];

    for (uint32_t ind = 0; ind < MaxFrequency(); ind++) {
        for (int i = 0; i < k; i++) {
            if ( freqOut->at(ind).real() > ordFreqOut->at(i)->at(1) ) {
```

```
            // Move back the rest
            for (int ii = k-1; ii > i; ii--){
               ordFreqOut->at(ii) = ordFreqOut->at(ii-1);
            }

            std::vector<double> * tV = new std::vector<double>(); tV->push_back(ind);
tV->push_back(freqOut->at(ind).real());
            ordFreqOut->at(i) = tV;
            break;
         }
      }
   }

   return ordFreqOut;
}

std::vector<double> * FFT::getNormalizedFrequency() {
   // Absolute Value EVERYTHING
   std::vector<double> * ABSFrequencyOutput = new std::vector<double>();
   double pHigh = -1;
   double pLow = -1;
   for (int i = 0; i < MaxFrequency(); i++) {
      double NormalizedValue = std::abs(FrequencyOutput->at(i).real());
      ABSFrequencyOutput->push_back(NormalizedValue);
      pHigh = std::max(pHigh, NormalizedValue);
      pLow = std::min(pLow, NormalizedValue);
   }


   std::vector<double> * NormalizedFrequencies = new std::vector<double>();
   for (int i = 0; i < MaxFrequency(); i++) {
      double NormalizedValue = ( ( 255.0 * ( ABSFrequencyOutput->at(i) - pLow ) ) / (
pHigh - pLow ) );
      NormalizedFrequencies->push_back(NormalizedValue);
      ValidateNormalizedPeak((double)i, NormalizedValue);
   }

   return NormalizedFrequencies;
}

void FFT::ValidatePeak(double ind, double p) {
   bool repl = false;
   for (int i = 0; i < k; i++) {
      if (p > peaks->at(i).at(1) && !repl) {

         for (int ii = k-2; ii >= i; ii--) {
            peaks->at(ii + 1).at(0) = peaks->at(ii).at(0);
            peaks->at(ii + 1).at(1) = peaks->at(ii).at(1);
```

```
            }

            peaks->at(i).at(0) = ind;
            peaks->at(i).at(1) = p;
            repl = true;
            break;
        }
    }
}

void FFT::ValidateNormalizedPeak(double ind, double p) {
    bool repl = false;
    for (int i = 0; i < k; i++) {

        std::vector<double> t1 = NormalizedPeaks->at(i);
        double t2 = NormalizedPeaks->at(i).at(0);

        if (p > NormalizedPeaks->at(i).at(1) && !repl) {

            for (int ii = k-2; ii >= i; ii--) {
                NormalizedPeaks->at(ii + 1).at(0) = NormalizedPeaks->at(ii).at(0);
                NormalizedPeaks->at(ii + 1).at(1) = NormalizedPeaks->at(ii).at(1);
            }

            NormalizedPeaks->at(i).at(0) = ind;
            NormalizedPeaks->at(i).at(1) = p;
            repl = true;
            break;
        }
    }
}

uint32_t FFT::MaxFrequency() {
    return std::min( FrequencyRange , SamplingRate/2 );
}
```

Note: this class uses a static sampling rate of 8000Hz, and as such we will preprocess all our files to this sampling rate.

Music Genre Classifier

Now to combine everything I researched into one product, I implemented a program which takes as input wav files of a few songs and their corresponding genres, and one wav file whose genre we want to predict, and outputs a guess of the genre of the prediction file.

The first thing we need is a dataset. So I wrote a small wrapper program which helps us create datasets on the fly, and test them.

CreateDataSet - Data Set Creator for Music Genre Classifier

CreateDataSet is a wrapper program for our actual classifier (ClassifyFrequencyArray.exe). CreateDataSet (CDS) allows us to create datasets, update existing datasets, preview them, and most importantly; test them.

CDS works on a directory, wherein all necessary files are placed, including temporary prediction files. There are four file types used by CDS: “vectors”, “labels”, “label_map”, “data_map”.

1. Vectors: A vectors file contains the actual data of a single or of multiple wav files. A vectors file has a header of three uint32_t in big endian: a magic number (2612), the sample count (how many vectors are in the file), and the vector size (currently 4,000).
2. Labels: A labels file contains the labels (uint8_t) corresponding to each vector in the vectors file. Label files begin with a two uint32_t header or a magic number (2211) and the sample count. The sample count written in the labels file and in the vectors file should be exactly the same. Notice that the labels and vectors files are codependent and must be written in the same order - the first label corresponds to the first vector, the second with the second, and so on. Therefore, they should be named the same (excluding the file extension) to keep track of which labels file corresponds to which vectors file.
3. Data Map: A data map file contains the file paths to the files already imported into the dataset. The data map file maps these file paths with the sound’s label. This file is useful when adding multiple files into the dataset as it keeps track of which files are already in the dataset and what their labels are. The format is such (Notice there is no header):

File Label (1 byte)	File Path (As null terminated string [ASCII only])
...	...
File Label (1 byte)	File Path (As null terminated string [ASCII only])

4. Label Map: A label map file, similarly to a data map file, contains information to make the program more user-friendly. The label map file of a dataset contains a map of each label - as uint8_t - to a literal label - as a 255 character (maximum) string. This file is used when showing the classes, or after a prediction is made, to convert

the uint8_t label into a human-legible and understandable string. Label map files also do not have a header.

We will also use CDS as a command-line tool to interface with the classifier (We will probably change this later, as this method wastes vast amounts of time loading and unloading data from files into memory).

Code:

Notice that although this is a C++ program, most of the code is in pure C as to increase performance (See Evolvement - Practical Research, where there is further explanation of the benefits of using C in C++).

CreateDataSet.cpp

```
#include <iostream>
#include <sys/stat.h>
#include <time.h>
#include <vector>
#include <map>

#include "AudioFile.h"
#include "FFT.h"

#include <direct.h>
#define GetCurrentDir _getcwd

#define MaximumLabelLength 255

std::string get_current_dir() {
   char buff[FILENAME_MAX];
   GetCurrentDir(buff, FILENAME_MAX);
   std::string current_working_dir(buff);
   return current_working_dir;
}
inline bool file_exists(const std::string& name) {
   struct stat buffer;
   return (stat(name.c_str(), &buffer) == 0);
}

int getFileSize(FILE * f) { // This will move the file pointer to SEEK_SET + 0
   fseek(f, 0, SEEK_END);
   int size = ftell(f);
   fseek(f, 0, SEEK_SET);
   return size;
}
```

```
uint32_t convert_to_big_endian(uint32_t b) {
    unsigned char bytes[4];
    memcpy(bytes, &b, sizeof(b));
    return (uint32_t)((bytes[0] << 24) | (bytes[1] << 16) | (bytes[2] << 8) | (bytes[3]));
}
uint32_t convert_to_little_endian(uint32_t b) {
    unsigned char bytes[4];
    memcpy(bytes, &b, sizeof(b));
    return (uint32_t)((bytes[0] << 24) | (bytes[1] << 16) | (bytes[2] << 8) | (bytes[3]));
}

// Change the last character of a string into a null byte (if it's a newline).
void fix_fgets(char * inp) {
    if (inp[strlen(inp) - 1] == '\n') { inp[strlen(inp) - 1] = 0x0; }
}

// Check if an input matches a command
bool cmpcommand(char * inp, const char * command) {
    if (memcmp(inp, command, strlen(command)) == 0) {
        char fin_char = inp[strlen(command)];
        // the format specified is that a command ends with one of the following four
characters.
        if (fin_char == ' ' || fin_char == '\t' || fin_char == '\n' || fin_char == '\0') {
            return true;
        }
    }
    return false;
}

// Global strings (character arrays)
char DataSetLabel[128] = { 0 };
char OutputPath[128] = { 0 };
char fn_label_map[128];
char fn_data_map[128];
char fn_vectors[128];
char fn_labels[128];

// Count the amount of arguments given in an input command (as ' ' delimiter)
int CountCommandArgs(char * command) {
    char * tmpCommand = (char *)malloc((strlen(command) + 1) * sizeof(char));
    memcpy(tmpCommand, command, strlen(command) + 1);

    int count = 0;
    const char delim[2] = " "; // delimiter
    char * next_tok;
    char * tok = strtok_s(tmpCommand, delim, &next_tok);
    while (tok != NULL) {
```

```
        printf("Arg #%d: '%s'\n", count, tok);
        count++;
        tok = strtok_s(NULL, delim, &next_tok);
    }

    free(tmpCommand);

    return count;
}

// Get command input argument at index as place it into output (also allocate memory for
output)
int GetCommandArg(char * command, int index, char ** output) {
    char * tmpCommand = (char *)malloc((strlen(command) + 1) * sizeof(char));
    memcpy(tmpCommand, command, strlen(command) + 1);

    int count = 0;
    const char delim[2] = " ";
    char * next_tok;
    char * tok = strtok_s(tmpCommand, delim, &next_tok);
    while (tok != NULL && count < index) {
        printf("Arg #%d: '%s'\n", count, tok);
        count++;
        tok = strtok_s(NULL, delim, &next_tok);
    }



    if (index > count) { free(tmpCommand); return -1; }

    *output = (char*)malloc((strlen(tok) + 1) * sizeof(char));
    memcpy(*output, tok, strlen(tok) + 1);

    free(tmpCommand);
    return 0;
}

// Update file paths using the user specified outputpath and the dataset global name
int UpdateFilePaths() {
    /*fn_label_map = (char*)calloc(128, sizeof(char));
    fn_data_map = (char*)calloc(128, sizeof(char));
    fn_vectors = (char*)calloc(128, sizeof(char));
    fn_labels = (char*)calloc(128, sizeof(char));*/

    sprintf_s(fn_label_map, 128, "%s/%s.label_map", OutputPath, DataSetLabel);
    sprintf_s(fn_data_map, 128, "%s/%s.data_map", OutputPath, DataSetLabel);
    sprintf_s(fn_vectors, 128, "%s/%s.vectors", OutputPath, DataSetLabel);
    sprintf_s(fn_labels, 128, "%s/%s.labels", OutputPath, DataSetLabel);
```

```
    return 0;
}
int ChangeOutputPath() {
    printf("# Please choose an output path for generated files.\n> ");
    memset(OutputPath, 0x0, 128 * sizeof(char));
    fgets(OutputPath, 127, stdin);
    fix_fgets(OutputPath);
    UpdateFilePaths();
    printf("# The data set will output to (vectors file example): '%s'\n", fn_vectors);
    return 0;
}
int ChangeDataSetName() {
    printf("# Please choose a name for this data set.\n> ");
    memset(DataSetLabel, 0x0, 128 * sizeof(char));
    fgets(DataSetLabel, 127, stdin);
    fix_fgets(DataSetLabel);
    UpdateFilePaths();
    printf("# The data set will output to (vectors file example): '%s'\n", fn_vectors);
    return 0;
}

// Count how many labels exist in the label map file.
int getExistingLabelsCount(uint8_t * existing_labels, FILE * fLabelMap) {
    int existing_labels_count = 0;
    char label_pair[MaximumLabelLength + 2];
    while (fread_s(label_pair, (MaximumLabelLength + 2) * sizeof(char), sizeof(char),
MaximumLabelLength + 1, fLabelMap)) {
        uint8_t label_key = label_pair[0];
        existing_labels[existing_labels_count] = label_key;
        existing_labels_count++;
    }
    return existing_labels_count;
}

// Prints the classes and their labels, which are stored in the label map file.
int ViewClassifications() {
    printf("# The following classifications exist in the data set: \n\n");

    uint8_t existing_labels[256];
    uint8_t existing_labels_count = 0;

    // A file with a map (dictionary) of the label numbers (0-255) and their corresponding
"human" definition.
    FILE * fLabelMap;
    errno_t errLabelMap;
    // A file with a map of the actual wav files which are in the vectors file.
    FILE * fDataMap;
```

```
    errno_t errDataMap;

    if (!file_exists(fn_label_map)) {
        printf("# As you can clearly see, there are none.\n");

        errLabelMap = fopen_s(&fLabelMap, fn_label_map, "wb");
        errDataMap = fopen_s(&fDataMap, fn_data_map, "wb");

        if (errDataMap || errLabelMap) { printf("# Could not open data or label map files!\n");
return -1; }
    }
    else {
        errLabelMap = fopen_s(&fLabelMap, fn_label_map, "r+b");
        errDataMap = fopen_s(&fDataMap, fn_data_map, "r+b");
        if (errDataMap || errLabelMap) { printf("# Could not open data or label map files!\n");
return -1; }

        fseek(fDataMap, 0, SEEK_END);

        /// Every 256bytes are one pair in the dictionary. Starting at byte 0.

        char label_pair[MaximumLabelLength + 2];
        while (fread_s(label_pair, (MaximumLabelLength + 2) * sizeof(char), sizeof(char),
MaximumLabelLength + 1, fLabelMap)) {
            uint8_t label_key = label_pair[0];
            printf("\t> %d | %s\n", label_key, label_pair + 1);
            existing_labels[existing_labels_count] = label_key;
            existing_labels_count++;
        }

        if (existing_labels_count == 0) {
            printf("# As you can clearly see, there are none.\n");
        }
    }

    fclose(fLabelMap);
    fclose(fDataMap);

    return 0;
}

// Prints the file paths of each file in the dataset sorted by their labels.
int ViewDataFiles() {
    printf("# The following files are in the data set: \n\n");

    uint8_t existing_labels[256];
    uint8_t existing_labels_count = 0;
```

```
   // A file with a map (dictionary) of the label numbers (0-255) and their corresponding
"human" definition.
   FILE * fLabelMap;
   errno_t errLabelMap;
   // A file with a map of the actual wav files which are in the vectors file.
   FILE * fDataMap;
   errno_t errDataMap;

   if (!file_exists(fn_data_map)) {
      printf("# As you can clearly see, there are none.\n");

      errLabelMap = fopen_s(&fLabelMap, fn_label_map, "wb");
      errDataMap = fopen_s(&fDataMap, fn_data_map, "wb");

      if (errDataMap || errLabelMap) { printf("# Could not open data or label map files!\n");
return -1; }
   }
   else {
      errLabelMap = fopen_s(&fLabelMap, fn_label_map, "r+b");
      errDataMap = fopen_s(&fDataMap, fn_data_map, "r+b");
      if (errDataMap || errLabelMap) { printf("# Could not open data or label map files!\n");
return -1; }

      int size = getFileSize(fDataMap); fseek(fDataMap, 0, SEEK_SET);
      char * file_data = (char*)malloc(size+1);
      int read_count = fread_s(file_data, size, sizeof(char), size, fDataMap);
      if (read_count == 0) {
         printf("# Error reading file.\n"); return -1;}


      std::map<uint8_t,char*> label_data_map;

      /// Every 256bytes are one pair in the dictionary. Starting at byte 0.

      char label_pair[MaximumLabelLength + 2];
      while (fread_s(label_pair, (MaximumLabelLength + 2) * sizeof(char), sizeof(char),
MaximumLabelLength + 1, fLabelMap)) {
         uint8_t label_key = label_pair[0];
         //existing_labels[existing_labels_count] = label_key;
         //label_data_map.insert(std::pair<uint8_t,char*>(label_key, label_pair + 1));

         printf("\t> %u | %s\n", label_key, label_pair + 1);

         char * live_data = file_data;

         int pos = 0;
         while (pos < size) {
            uint8_t data_key = live_data[pos];
```

```
            if (data_key == label_key) {
               printf("\t|--> %s\n", live_data + pos + 1);
            }
            pos += 2 + strlen(live_data + pos + 1);
         }

         existing_labels_count++;
      }

      if (existing_labels_count == 0) {
         printf("# As you can clearly see, there are none.\n");
      }
      /*else {

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

            uint8_t key = existing_labels[i];
            printf("\t> %u | %s\n", key, label_data_map.at(key));

            char * live_data = file_data;

            int pos = 0;
            while (pos < size) {
               uint8_t data_key = live_data[pos];
               if (data_key == key) {
                  printf("\t|--> %s\n", live_data + pos + 1);
               }
               pos += 2 + strlen(live_data + pos + 1);
            }
         }

      }*/

      free(file_data);
   }

   fclose(fLabelMap);
   fclose(fDataMap);

   return 0;
}
int AddDataToSet() {

   std::string WAV_File_Path;
   std::cout << "# Path for WAV file: " << std::endl;
   std::cout << get_current_dir() << "\\";
   WAV_File_Path = get_current_dir() + "\\";
   char WAV_File_Path_buffer[256];
```

```
    fgets(WAV_File_Path_buffer, sizeof(WAV_File_Path_buffer), stdin);
    fix_fgets(WAV_File_Path_buffer);
    for (int i = 0; i < strlen(WAV_File_Path_buffer); i++) {
        WAV_File_Path += WAV_File_Path_buffer[i];
    }
    std::cout << std::endl << std::endl;

    AudioFile<double> * audioFile = new AudioFile<double>();

    if (audioFile->load(WAV_File_Path)) {
        std::cout << "# Loaded File!" << std::endl;
        audioFile->printSummary();
        std::cout << "# CH: " << audioFile->getNumChannels() << " | " << "SAMP: " <<
audioFile->getNumSamplesPerChannel() * audioFile->getNumChannels() << std::endl;

        audioFile->ConcactChannels();

        FFT * FT = new FFT();

        uint8_t channelInd = 0;
        for (int sampInd = 0; sampInd < audioFile->getNumSamplesPerChannel();
sampInd++) {
            FT->AppendToWave(audioFile->samples[channelInd][sampInd]);
        }
        int actInd = 0;
        std::cout << "\t" << ++actInd << ") " << "Loaded \"" << WAV_File_Path << "\" into
Fourier Transform!" << std::endl;

        uint32_t sample_count = audioFile->getNumSamplesPerChannel() /
audioFile->getSampleRate();

        // A file with a map (dictionary) of the label numbers (0-255) and their corresponding
"human" definition.
        FILE * fLabelMap;
        // A file with a map of the actual wav files which are in the vectors file.
        FILE * fDataMap;

        errno_t errLabelMap = fopen_s(&fLabelMap, fn_label_map, "r+b");
        errno_t errDataMap = fopen_s(&fDataMap, fn_data_map, "r+b");

        if (errDataMap || errLabelMap) { printf("# Could not open Data or Label map
files!\n"); return -1; }

        // Create new vector and label files, or use existing ones and change the sample count
        FILE * fVect;
        FILE * fLabel;
        if (!file_exists(fn_vectors)) {
            errno_t errVects = fopen_s(&fVect, fn_vectors, "wb");
```

```
        errno_t errLabels = fopen_s(&fLabel, fn_labels, "wb");
        uint32_t intro[3] = { convert_to_big_endian(2612),
convert_to_big_endian(sample_count), convert_to_big_endian(4000) };
        uint32_t introLabels[2] = { convert_to_big_endian(2211),
convert_to_big_endian(sample_count) };
        fwrite((char*)intro, sizeof(char), 4 * sizeof(uint32_t), fVect);
        fwrite((char*)introLabels, sizeof(char), 2 * sizeof(uint32_t), fLabel);
    }
    else {
        errno_t errVects = fopen_s(&fVect, fn_vectors, "r+b");
        errno_t errLabels = fopen_s(&fLabel, fn_labels, "r+b");
        uint32_t old_sample_count = 0;
        fseek(fLabel, sizeof(uint32_t), SEEK_SET);
        fread_s(&old_sample_count, sizeof(old_sample_count), sizeof(uint32_t), 1,
fLabel);
        old_sample_count = convert_to_little_endian(old_sample_count);
        old_sample_count += sample_count;
        old_sample_count = convert_to_big_endian(old_sample_count);
        fseek(fLabel, sizeof(uint32_t), SEEK_SET);
        fseek(fVect, sizeof(uint32_t), SEEK_SET);
        fwrite(&old_sample_count, sizeof(uint32_t), 1, fLabel);
        fwrite(&old_sample_count, sizeof(uint32_t), 1, fVect);
        fseek(fLabel, 0, SEEK_END);
        fseek(fVect, 0, SEEK_END);
    }


    printf("Please enter the label (as uint8) for this sound.\n> ");
    char label_buffer[16];
    fgets(label_buffer, sizeof(label_buffer), stdin);
    fix_fgets(label_buffer);

    long lab_long = strtol(label_buffer, NULL, 10);

    uint8_t lab = 0;
    memcpy(&lab, &lab_long, sizeof(uint8_t));
    char label[1] = { lab };


    /*char * label_buffer_temp = (char*)malloc(strlen(label_buffer));
    memcpy(label_buffer_temp, label_buffer, strlen(label_buffer));
    printf("\n\nGot label! %u\n\n", label_buffer_temp[0]);

    sprintf_s(label, sizeof(label), "%u", label_buffer_temp[0]);

    free(label_buffer_temp);*/

    uint8_t existing_labels[256];
```

```
        int existing_labels_count = getExistingLabelsCount(existing_labels, fLabelMap);

        fseek(fDataMap, 0, SEEK_END);
        fseek(fLabelMap, 0, SEEK_END);

        fwrite(label, sizeof(uint8_t), 1, fDataMap);
        /*char * WAV_File_Path_c_str = (char*)calloc(WAV_File_Path.length(),
sizeof(char));
        strncpy_s(WAV_File_Path_c_str, WAV_File_Path.length(), WAV_File_Path.c_str(),
WAV_File_Path.length());
        fwrite(WAV_File_Path_c_str, sizeof(char), WAV_File_Path.length(), fDataMap);*/
        fwrite(WAV_File_Path.c_str(), sizeof(char), WAV_File_Path.length(), fDataMap);
        fwrite(OutputPath + 127, sizeof(char), 1, fDataMap);

        bool exists = false;
        if (existing_labels_count > 0) {

            int i = 0;
            while (i < 256 && !exists) {
                exists = exists || existing_labels[i] == lab;
                i++;
            }
        }
        if (!exists) {
            printf("# This is a new label, please describe it. (in 255 or less characters)\n> ");
            char label_description_buffer[MaximumLabelLength + 1];
            fgets(label_description_buffer, MaximumLabelLength, stdin);
            fix_fgets(label_description_buffer);

            fwrite(label, 1, 1, fLabelMap);
            fwrite(label_description_buffer, sizeof(char), MaximumLabelLength, fLabelMap);

            printf("# Added the pair: {%u, %s} to the label_map.\n", lab,
label_description_buffer);
        }

        printf("# Now loading the file with the label %d.\n", lab);

        /// Create an array of the frequencies
        for (int x = 0; x < sample_count; x++) {

            std::vector<Complex> * liveFreq =
FT->FourierTransfer_Part(audioFile->getSampleRate(), x);

            char * freqArr = (char *)calloc(FT->MaxFrequency(), sizeof(char));

            for (int freqInd = 0; freqInd < FT->MaxFrequency(); freqInd++) {
                freqArr[freqInd] = liveFreq->at(freqInd).real();
```

```
            }

            fwrite(freqArr, sizeof(char), FT->MaxFrequency(), fVect);
            fwrite(label, sizeof(char), 1, fLabel);
            free(freqArr);
        }
        fclose(fVect);
        fclose(fLabel);
        fclose(fLabelMap);
        fclose(fDataMap);

        delete FT;

        printf("# Appended sound!\n");
    }

    delete audioFile;

    return 0;
}

// Initialize a new dataset
int Initialize() {
    FILE * fLabelMap;
    errno_t errLabelMap;
    FILE * fDataMap;
    errno_t errDataMap;
    errLabelMap = fopen_s(&fLabelMap, fn_label_map, "wb");
    errDataMap = fopen_s(&fDataMap, fn_data_map, "wb");
    fclose(fLabelMap);
    fclose(fDataMap);

    return 0;
}

// predict the label of a file using the Classifier
/*
    The predict function can take arguments such as:
    1) a file path.
        The first input can be the relative path to a vectors file which you want to use instead
of processing a wav file into a vectors file.
    2) the '-f' argument.
        This is passed into the classifier as 'fast' which tells the program to only check some of
the data points (arbitrarily) and not all of them.
        This obviously results in faster execution time (10x faster - since we take only a tenth
of the vectors), but there is a penalty to the accuracy of the results. Though the final
prediction is still usually correct.
*/
```

```
int Predict(char * command) {
    int argc = CountCommandArgs(command);
    printf("Arg count: %d\n", argc);
    if (argc == 0) {
        char predictOutputFile[256];
        char predictOutputFile_tmp[128];
        printf("Name your prediction file: \n> ");
        fgets(predictOutputFile_tmp, sizeof(predictOutputFile_tmp), stdin);
        fix_fgets(predictOutputFile_tmp);
        sprintf_s(predictOutputFile, sizeof(predictOutputFile), "%s/%s.vectors", OutputPath,
predictOutputFile_tmp);

        std::string WAV_File_Path;
        std::cout << "# Path for WAV file: " << std::endl;
        std::cout << get_current_dir() << "\\";
        WAV_File_Path = get_current_dir() + "\\";
        char WAV_File_Path_buffer[256];
        fgets(WAV_File_Path_buffer, sizeof(WAV_File_Path_buffer), stdin);
        fix_fgets(WAV_File_Path_buffer);
        for (int i = 0; i < strlen(WAV_File_Path_buffer); i++) {
            WAV_File_Path += WAV_File_Path_buffer[i];
        }
        std::cout << std::endl << std::endl;

        AudioFile<double> * audioFile = new AudioFile<double>();

        // Parse audio file into FFT and export it as vectors into a vectors file
        if (audioFile->load(WAV_File_Path)) {
            std::cout << "# Loaded File!" << std::endl;
            audioFile->printSummary();
            std::cout << "# CH: " << audioFile->getNumChannels() << " | " << "SAMP: " <<
audioFile->getNumSamplesPerChannel() * audioFile->getNumChannels() << std::endl;

            FFT * FT = new FFT();

            uint8_t channelInd = 0;
            for (int sampInd = 0; sampInd < audioFile->getNumSamplesPerChannel();
sampInd++) {
                FT->AppendToWave(audioFile->samples[channelInd][sampInd]);
            }
            int actInd = 0;
            std::cout << "\t" << ++actInd << ") " << "Loaded \"" << WAV_File_Path << "\"
into Fourier Transform!" << std::endl;

            uint32_t sample_count = audioFile->getNumSamplesPerChannel() /
audioFile->getSampleRate();

            FILE * fVect;
```

```
        errno_t errVects;
        if (!file_exists(predictOutputFile)) {
            errVects = fopen_s(&fVect, predictOutputFile, "wb");
            uint32_t intro[3] = { convert_to_big_endian(2612),
convert_to_big_endian(sample_count), convert_to_big_endian(4000) };
            fwrite((char*)intro, sizeof(char), 3 * sizeof(uint32_t), fVect);
        }
        else {
            errVects = fopen_s(&fVect, predictOutputFile, "r+b");
            uint32_t old_sample_count = 0;
            fread_s(&old_sample_count, sizeof(old_sample_count), sizeof(uint32_t), 1,
fVect);
            old_sample_count = convert_to_little_endian(old_sample_count);
            old_sample_count += sample_count;
            old_sample_count = convert_to_big_endian(old_sample_count);
            fseek(fVect, sizeof(uint32_t), SEEK_SET);
            fwrite(&old_sample_count, sizeof(uint32_t), 1, fVect);
            fseek(fVect, 0, SEEK_END);
        }
        if (errVects) { printf("Could not find nor create prediction file!\n"); return -1; }

        /// Create an array of the frequencies
        for (int x = 0; x < sample_count; x++) {

            std::vector<Complex> * liveFreq =
FT->FourierTransfer_Part(audioFile->getSampleRate(), x);

            char * freqArr = (char *)calloc(FT->MaxFrequency(), sizeof(char));

            for (int freqInd = 0; freqInd < FT->MaxFrequency(); freqInd++) {
                freqArr[freqInd] = liveFreq->at(freqInd).real();
            }

            fwrite(freqArr, sizeof(char), FT->MaxFrequency(), fVect);
            free(freqArr);
        }
        fclose(fVect);

        delete FT;
    }
    delete audioFile;

    printf("# Created prediction file!!\n");


    char path[2048];
    sprintf_s(path, sizeof(path), "ClassifyFrequencyArray.exe \"%s\" \"%s\" \"%s\"",
OutputPath, DataSetLabel, predictOutputFile);
```

```
        int prediction_output = system(path);

        FILE * fLabelMap;
        errno_t errLabelMap = fopen_s(&fLabelMap, fn_label_map, "r+b");
        if (errLabelMap) { printf("# Could not open label map file!\n"); return -1; }

        char label_pair[MaximumLabelLength + 2];
        while (fread_s(label_pair, (MaximumLabelLength + 2) * sizeof(char), sizeof(char),
MaximumLabelLength + 1, fLabelMap)) {
            uint8_t label_key = label_pair[0];
            if (label_key == prediction_output) {
                printf("The prediction for this file is: \n\t%d | %s\n", prediction_output,
label_pair + 1);
            }
        }

        fclose(fLabelMap);
    }
    else if (argc == 1) {
        char * predictOutputFile;
        if (GetCommandArg(command, 0, &predictOutputFile) != 0) { printf("# Could not
find argument!\n"); return -1; }
        char path[2048];
        sprintf_s(path, sizeof(path), "ClassifyFrequencyArray.exe \"%s\" \"%s\" \"%s\"",
OutputPath, DataSetLabel, predictOutputFile);

        int prediction_output = system(path);

        FILE * fLabelMap;
        errno_t errLabelMap = fopen_s(&fLabelMap, fn_label_map, "r+b");
        if (errLabelMap) { printf("# Could not open label map file!\n"); return -1; }

        char label_pair[MaximumLabelLength + 2];
        while (fread_s(label_pair, (MaximumLabelLength + 2) * sizeof(char), sizeof(char),
MaximumLabelLength + 1, fLabelMap)) {
            uint8_t label_key = label_pair[0];
            if (label_key == prediction_output) {
                printf("The prediction for this file is: \n\t%d | %s\n", prediction_output,
label_pair + 1);
            }
        }

        fclose(fLabelMap);
    }
    else if (argc == 2) {
        char * optionCommand;
        if (GetCommandArg(command, 0, &optionCommand) != 0) { printf("# Could not
```

```
find argument!\n"); return -1; }
        if (strcmp(optionCommand, "-f") == 0) {
            char * predictOutputFile;
            if (GetCommandArg(command, 1, &predictOutputFile) != 0) { printf("# Could not
find argument!\n"); return -1; }
            char path[2048];
            sprintf_s(path, sizeof(path), "ClassifyFrequencyArray.exe \"%s\" \"%s\" \"%s\"
\"fast\"", OutputPath, DataSetLabel, predictOutputFile);

            int prediction_output = system(path);

            FILE * fLabelMap;
            errno_t errLabelMap = fopen_s(&fLabelMap, fn_label_map, "r+b");
            if (errLabelMap) { printf("# Could not open label map file!\n"); return -1; }

            char label_pair[MaximumLabelLength + 2];
            while (fread_s(label_pair, (MaximumLabelLength + 2) * sizeof(char), sizeof(char),
MaximumLabelLength + 1, fLabelMap)) {
                uint8_t label_key = label_pair[0];
                if (label_key == prediction_output) {
                    printf("The prediction for this file is: \n\t%d | %s\n", prediction_output,
label_pair + 1);
                }
            }
            fclose(fLabelMap);
        }
    }
    else {
    printf("# Unrecognized argument count!\n");
    }
    return 0;
}
int ExitProgram() {
    free(fn_data_map);
    free(fn_label_map);
    free(fn_labels);
    free(fn_vectors);
    exit(0);
    return 0;
}
int Help(bool ext) {
    printf("You can use the following commands:\n");
    printf("\t> help\t\t\t| To see this menu.\n");
    printf("\t> help extended\t\t| To see a full list of commands.\n");
    printf("\t> init\t\t\t| Initialize dataset files. You must do this first when creating a new
dataset!\n");
    printf("\t> change data set name\t| To change the name given to output files.\n");
    if (ext) {
```

```
        printf("\t> change dataset name\t| To change the name given to output files.\n");
        printf("\t> change data name\t| To change the name given to output files.\n");
    }
    printf("\t> change output\t\t| To change the output path of created files.\n");
    printf("\t> list classes\t\t| To see which classifications exist in the data set.\n");
    if (ext) {
        printf("\t> list classifications\t| To see which classifications exist in the data set.\n");
        printf("\t> list class\t\t| To see which classifications exist in the data set.\n");
        printf("\t> view classes\t\t| To see which classifications exist in the data set.\n");
        printf("\t> view classifications\t| To see which classifications exist in the data set.\n");
        printf("\t> view class\t\t| To see which classifications exist in the data set.\n");
    }
    printf("\t> list files\t\t| To see a sorted list of the files which have been loaded into the
dataset.\n");
    if (ext) {
        printf("\t> list data\t\t| To see a sorted list of the files which have been loaded into the
dataset.\n");
        printf("\t> view files\t\t| To see a sorted list of the files which have been loaded into
the dataset.\n");
        printf("\t> view data\t\t| To see a sorted list of the files which have been loaded into
the dataset.\n");
    }
    printf("\t> add file\t\t| To add another file into the dataset.\n");
    if (ext) {
        printf("\t> add data\t\t| To add another file into the dataset.\n");
    }
    printf("\t> predict\t\t| Predict the genre of a wav file.\n");
    printf("\t> exit\t\t\t| To close the program.\n");

    return 0;
}

// Gets user input and compares it to valid commands
int ProccessCommands() {
    printf("> ");
    char input[1024];
    fgets(input, sizeof(input), stdin);
    fix_fgets(input);

    if (cmpcommand(input, "help") || cmpcommand(input, "help extended") ||
cmpcommand(input, "help ext")) { return Help(cmpcommand(input, "help extended") ||
cmpcommand(input, "help ext")); }
    if (cmpcommand(input, "change output")) { return ChangeOutputPath(); }
    if (cmpcommand(input, "change data set name") || cmpcommand(input, "change dataset
name") || cmpcommand(input, "change data name")) { return ChangeOutputPath(); }
    if (cmpcommand(input, "list classifications") || cmpcommand(input, "view
classifications") || cmpcommand(input, "view class") || cmpcommand(input, "list class") ||
cmpcommand(input, "view classes") || cmpcommand(input, "list classes")) { return
```

```
ViewClassifications(); }
   if (cmpcommand(input, "list files") || cmpcommand(input, "list data") ||
cmpcommand(input, "view files") || cmpcommand(input, "view data")) { return
ViewDataFiles(); }
   if (cmpcommand(input, "add file") || cmpcommand(input, "add data")) { return
AddDataToSet(); }
   if (cmpcommand(input, "predict")) { return Predict(input + strlen("predict") + 1); }
   if (cmpcommand(input, "init") || cmpcommand(input, "initialize")) { return Initialize(); }
   if (cmpcommand(input, "exit") || cmpcommand(input, "close")) { return ExitProgram();
}

   return 1;
}

int main()
{
   std::cout << "=== === === === ===" << std::endl;
   std::cout << "Started WAV Analyzer" << std::endl;
   std::cout << "  Data Set Editor" << std::endl;
   std::cout << "=== === === === ===" << std::endl;
   std::cout << std::endl << std::endl;

   ChangeOutputPath();
   ChangeDataSetName();

   while (true) {
      if (ProccessCommands() == 1) {
         printf("# Command Failed!\n");
      }
   }

   return 0;
}
```

AudioFile.h

Other than CreateDataSet.cpp, and obviously the FFT class we wrote earlier, there is one more file included in CDS. This header file, which is the only part of the project which was not written from scratch. The code loads a Wave (WAV) or Audio Interchange File Format (AIFF) file from the file system and allows us to directly interface with the sample array. This sample array is actually the input for our Fourier Transform, as it literally is samples of the original sound wave.

The code for this file was generously provided by Adam Stark under the GNU General Public License. Link to original publication: https://github.com/adamstark/AudioFile

```
#pragma once
//=============================================================================
=======
/** @file AudioFile.h
 *  @author Adam Stark
 *  @copyright Copyright (C) 2017  Adam Stark
 *
 * This file is part of the 'AudioFile' library
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied  warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.   See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */

//=============================================================================
=======


//=============================================================================
=======
/**
*  This file has been modified by Michael K. Steinberg (2021)
*
*  Many, many thanks to Adam Stark for the original code! :)
*/
//=============================================================================
```

```
=======

#ifndef _AS_AudioFile_h
#define _AS_AudioFile_h

#include <iostream>
#include <vector>
#include <assert.h>
#include <string>
#include <fstream>
#include <unordered_map>
#include <iterator>
#include <algorithm>
#include <cstring>

// disable some warnings on Windows
#if defined (_MSC_VER)
__pragma(warning(push))
__pragma(warning(disable : 4244))
__pragma(warning(disable : 4457))
__pragma(warning(disable : 4458))
__pragma(warning(disable : 4389))
__pragma(warning(disable : 4996))
#elif defined (__GNUC__)
_Pragma("GCC diagnostic push")
_Pragma("GCC diagnostic ignored \"-Wconversion\"")
_Pragma("GCC diagnostic ignored \"-Wsign-compare\"")
_Pragma("GCC diagnostic ignored \"-Wshadow\"")
#endif

//=============================================================
/** The different types of audio file, plus some other types to
 * indicate a failure to load a file, or that one hasn't been
 * loaded yet
 */
   enum class AudioFileFormat
{
   Error,
   NotLoaded,
   Wave,
   Aiff
};

//=============================================================
template <class T>
class AudioFile
{
public:
```

```
    //==============================================================
    typedef std::vector<std::vector<T> > AudioBuffer;

    //==============================================================
    /** Constructor */
    AudioFile();

    //==============================================================
    /** Loads an audio file from a given file path.
     * @Returns true if the file was successfully loaded
     */
    bool load(std::string filePath);

    /** Saves an audio file to a given file path.
     * @Returns true if the file was successfully saved
     */
    bool save(std::string filePath, AudioFileFormat format = AudioFileFormat::Wave);

    //==============================================================
    /** @Returns the sample rate */
    uint32_t getSampleRate() const;

    /** @Returns the number of audio channels in the buffer */
    int getNumChannels() const;

    /** @Returns true if the audio file is mono */
    bool isMono() const;

    /** @Returns true if the audio file is stereo */
    bool isStereo() const;

    /** @Returns the bit depth of each sample */
    int getBitDepth() const;

    /** @Returns the number of samples per channel */
    int getNumSamplesPerChannel() const;

    /** @Returns the length in seconds of the audio file based on the number of samples and
sample rate */
    double getLengthInSeconds() const;

    /** Prints a summary of the audio file to the console */
    void printSummary() const;

    //==============================================================

    /** Set the audio buffer for this AudioFile by copying samples from another buffer.
```

```
    * @Returns true if the buffer was copied successfully.
    */
    bool setAudioBuffer(AudioBuffer& newBuffer);

    /** Sets the audio buffer to a given number of channels and number of samples per channel. This will try to preserve
     * the existing audio, adding zeros to any new channels or new samples in a given channel.
     */
    void setAudioBufferSize(int numChannels, int numSamples);

    /** Sets the number of samples per channel in the audio buffer. This will try to preserve
     * the existing audio, adding zeros to new samples in a given channel if the number of samples is increased.
     */
    void setNumSamplesPerChannel(int numSamples);

    /** Sets the number of channels. New channels will have the correct number of samples and be initialised to zero */
    void setNumChannels(int numChannels);

    /** Sets the bit depth for the audio file. If you use the save() function, this bit depth rate will be used */
    void setBitDepth(int numBitsPerSample);

    /** Sets the sample rate for the audio file. If you use the save() function, this sample rate will be used */
    void setSampleRate(uint32_t newSampleRate);

    //==============================================================
    /** Sets whether the library should log error messages to the console. By default this is true */
    void shouldLogErrorsToConsole(bool logErrors);

    //==============================================================
    /** A vector of vectors holding the audio samples for the AudioFile. You can
     * access the samples by channel and then by sample index, i.e:
     *
     *      samples[channel][sampleIndex]
     */
    AudioBuffer samples;

    //==============================================================
    /** An optional iXML chunk that can be added to the AudioFile. */
    std::string iXMLChunk;

    //==============================================================
    /** Concact all channels into channel 0 */
```

```
    void ConcactChannels();

private:

    //=============================================================
    enum class Endianness
    {
        LittleEndian,
        BigEndian
    };

    //=============================================================
    AudioFileFormat determineAudioFileFormat(std::vector<uint8_t>& fileData);
    bool decodeWaveFile(std::vector<uint8_t>& fileData);
    bool decodeAiffFile(std::vector<uint8_t>& fileData);

    //=============================================================
    void clearAudioBuffer();

    //=============================================================
    int32_t fourBytesToInt(std::vector<uint8_t>& source, int startIndex, Endianness
endianness = Endianness::LittleEndian);
    int16_t twoBytesToInt(std::vector<uint8_t>& source, int startIndex, Endianness
endianness = Endianness::LittleEndian);
    int getIndexOfString(std::vector<uint8_t>& source, std::string s);
    int getIndexOfChunk(std::vector<uint8_t>& source, const std::string& chunkHeaderID,
int startIndex, Endianness endianness = Endianness::LittleEndian);

    //=============================================================
    T sixteenBitIntToSample(int16_t sample);
    int16_t sampleToSixteenBitInt(T sample);

    //=============================================================
    uint8_t sampleToSingleByte(T sample);
    T singleByteToSample(uint8_t sample);

    uint32_t getAiffSampleRate(std::vector<uint8_t>& fileData, int sampleRateStartIndex);
    bool tenByteMatch(std::vector<uint8_t>& v1, int startIndex1, std::vector<uint8_t>& v2,
int startIndex2);
    void addSampleRateToAiffData(std::vector<uint8_t>& fileData, uint32_t sampleRate);
    T clamp(T v1, T minValue, T maxValue);

    //=============================================================
    void reportError(std::string errorMessage);

    //=============================================================
    AudioFileFormat audioFileFormat;
    uint32_t sampleRate;
```

```
    int bitDepth;
    bool logErrorsToConsole{ true };
};


//==========================================================
// Pre-defined 10-byte representations of common sample rates
static std::unordered_map <uint32_t, std::vector<uint8_t>> aiffSampleRateTable = {
    {8000, {64, 11, 250, 0, 0, 0, 0, 0, 0, 0}},
    {11025, {64, 12, 172, 68, 0, 0, 0, 0, 0, 0}},
    {16000, {64, 12, 250, 0, 0, 0, 0, 0, 0, 0}},
    {22050, {64, 13, 172, 68, 0, 0, 0, 0, 0, 0}},
    {32000, {64, 13, 250, 0, 0, 0, 0, 0, 0, 0}},
    {37800, {64, 14, 147, 168, 0, 0, 0, 0, 0, 0}},
    {44056, {64, 14, 172, 24, 0, 0, 0, 0, 0, 0}},
    {44100, {64, 14, 172, 68, 0, 0, 0, 0, 0, 0}},
    {47250, {64, 14, 184, 146, 0, 0, 0, 0, 0, 0}},
    {48000, {64, 14, 187, 128, 0, 0, 0, 0, 0, 0}},
    {50000, {64, 14, 195, 80, 0, 0, 0, 0, 0, 0}},
    {50400, {64, 14, 196, 224, 0, 0, 0, 0, 0, 0}},
    {88200, {64, 15, 172, 68, 0, 0, 0, 0, 0, 0}},
    {96000, {64, 15, 187, 128, 0, 0, 0, 0, 0, 0}},
    {176400, {64, 16, 172, 68, 0, 0, 0, 0, 0, 0}},
    {192000, {64, 16, 187, 128, 0, 0, 0, 0, 0, 0}},
    {352800, {64, 17, 172, 68, 0, 0, 0, 0, 0, 0}},
    {2822400, {64, 20, 172, 68, 0, 0, 0, 0, 0, 0}},
    {5644800, {64, 21, 172, 68, 0, 0, 0, 0, 0, 0}}
};

//==========================================================
enum WavAudioFormat
{
    PCM = 0x0001,
    IEEEFloat = 0x0003,
    ALaw = 0x0006,
    MULaw = 0x0007,
    Extensible = 0xFFFE
};

//==========================================================
enum AIFFAudioFormat
{
    Uncompressed,
    Compressed,
    Error
};

//==========================================================
```

```
/* IMPLEMENTATION */
//=============================================================

//=============================================================
template <class T>
AudioFile<T>::AudioFile()
{
    static_assert(std::is_floating_point<T>::value, "Error: This version of AudioFile only supports floating point sample formats.");

    bitDepth = 16;
    sampleRate = 44100;
    samples.resize(1);
    samples[0].resize(0);
    audioFileFormat = AudioFileFormat::NotLoaded;
}

//=============================================================
template <class T>
uint32_t AudioFile<T>::getSampleRate() const
{
    return sampleRate;
}

//=============================================================
template <class T>
int AudioFile<T>::getNumChannels() const
{
    return (int)samples.size();
}

//=============================================================
template <class T>
bool AudioFile<T>::isMono() const
{
    return getNumChannels() == 1;
}

//=============================================================
template <class T>
bool AudioFile<T>::isStereo() const
{
    return getNumChannels() == 2;
}

//=============================================================
template <class T>
int AudioFile<T>::getBitDepth() const
```

```
{
    return bitDepth;
}

//=============================================================
template <class T>
int AudioFile<T>::getNumSamplesPerChannel() const
{
    if (samples.size() > 0) {
        return (int)samples[0].size();
    } else {
        return 0;
    }
}

//=============================================================
template <class T>
double AudioFile<T>::getLengthInSeconds() const
{
    return (double)getNumSamplesPerChannel() / (double)sampleRate;
}

//=============================================================
template <class T>
void AudioFile<T>::printSummary() const
{
    std::cout << "|======================================|" << std::endl;
    std::cout << "Number of Channels: " << getNumChannels() << std::endl;
    std::cout << "Number of Samples per Channel: " << getNumSamplesPerChannel() <<
std::endl;
    std::cout << "Sample Rate: " << sampleRate << std::endl;
    std::cout << "Bit Depth: " << bitDepth << std::endl;
    std::cout << "Length in Seconds: " << getLengthInSeconds() << std::endl;
    std::cout << "|======================================|" << std::endl;
}

//=============================================================
template <class T>
bool AudioFile<T>::setAudioBuffer(AudioBuffer& newBuffer)
{
    int numChannels = (int)newBuffer.size();

    if (numChannels <= 0)
    {
        assert(false && "The buffer your are trying to use has no channels.");
        return false;
    }
```

```
    size_t numSamples = newBuffer[0].size();

    // set the number of channels
    samples.resize(newBuffer.size());

    for (int k = 0; k < getNumChannels(); k++)
    {
        assert(newBuffer[k].size() == numSamples);

        samples[k].resize(numSamples);

        for (size_t i = 0; i < numSamples; i++)
        {
            samples[k][i] = newBuffer[k][i];
        }
    }

    return true;
}

//=============================================================
template <class T>
void AudioFile<T>::setAudioBufferSize(int numChannels, int numSamples)
{
    samples.resize(numChannels);
    setNumSamplesPerChannel(numSamples);
}

//=============================================================
template <class T>
void AudioFile<T>::setNumSamplesPerChannel(int numSamples)
{
    int originalSize = getNumSamplesPerChannel();

    for (int i = 0; i < getNumChannels(); i++)
    {
        samples[i].resize(numSamples);

        // set any new samples to zero
        if (numSamples > originalSize)
            std::fill(samples[i].begin() + originalSize, samples[i].end(), (T)0.);
    }
}

//=============================================================
template <class T>
void AudioFile<T>::setNumChannels(int numChannels)
{
```

```
    int originalNumChannels = getNumChannels();
    int originalNumSamplesPerChannel = getNumSamplesPerChannel();

    samples.resize(numChannels);

    // make sure any new channels are set to the right size
    // and filled with zeros
    if (numChannels > originalNumChannels)
    {
        for (int i = originalNumChannels; i < numChannels; i++)
        {
            samples[i].resize(originalNumSamplesPerChannel);
            std::fill(samples[i].begin(), samples[i].end(), (T)0.);
        }
    }
}

//=============================================================
template <class T>
void AudioFile<T>::setBitDepth(int numBitsPerSample)
{
    bitDepth = numBitsPerSample;
}

//=============================================================
template <class T>
void AudioFile<T>::setSampleRate(uint32_t newSampleRate)
{
    sampleRate = newSampleRate;
}

//=============================================================
template <class T>
void AudioFile<T>::shouldLogErrorsToConsole(bool logErrors)
{
    logErrorsToConsole = logErrors;
}

//=============================================================
template <class T>
bool AudioFile<T>::load(std::string filePath)
{
    std::ifstream file(filePath, std::ios::binary);

    // check the file exists
    if (!file.good())
    {
        reportError("Error: This file doesn't exist or otherwise can't be loaded.\n" + filePath);
```

```
        return false;
    }

    file.unsetf(std::ios::skipws);
    std::istream_iterator<uint8_t> begin(file), end;
    std::vector<uint8_t> fileData(begin, end);

    // get audio file format
    audioFileFormat = determineAudioFileFormat(fileData);

    if (audioFileFormat == AudioFileFormat::Wave)
    {
        return decodeWaveFile(fileData);
    }
    else if (audioFileFormat == AudioFileFormat::Aiff)
    {
        return decodeAiffFile(fileData);
    }
    else
    {
        reportError("Audio File Type: Error");
        return false;
    }
}

//=============================================================
template <class T>
bool AudioFile<T>::decodeWaveFile(std::vector<uint8_t>& fileData)
{
    // -------------------------------------------------------
    // HEADER CHUNK
    std::string headerChunkID(fileData.begin(), fileData.begin() + 4);
    //int32_t fileSizeInBytes = fourBytesToInt (fileData, 4) + 8;
    std::string format(fileData.begin() + 8, fileData.begin() + 12);

    // -------------------------------------------------------
    // try and find the start points of key chunks
    int indexOfDataChunk = getIndexOfChunk(fileData, "data", 12);
    int indexOfFormatChunk = getIndexOfChunk(fileData, "fmt ", 12);
    int indexOfXMLChunk = getIndexOfChunk(fileData, "iXML", 12);

    // if we can't find the data or format chunks, or the IDs/formats don't seem to be as
expected
    // then it is unlikely we'll able to read this file, so abort
    if (indexOfDataChunk == -1 || indexOfFormatChunk == -1 || headerChunkID != "RIFF"
|| format != "WAVE")
    {
        reportError("Error: This doesn't seem to be a valid .WAV file!\n");
```

```
        return false;
    }

    // ----------------------------------------------------------
    // FORMAT CHUNK
    int f = indexOfFormatChunk;
    std::string formatChunkID(fileData.begin() + f, fileData.begin() + f + 4);
    //int32_t formatChunkSize = fourBytesToInt (fileData, f + 4);
    int16_t audioFormat = twoBytesToInt(fileData, f + 8);
    int16_t numChannels = twoBytesToInt(fileData, f + 10);
    sampleRate = (uint32_t)fourBytesToInt(fileData, f + 12);
    int32_t numBytesPerSecond = fourBytesToInt(fileData, f + 16);
    int16_t numBytesPerBlock = twoBytesToInt(fileData, f + 20);
    bitDepth = (int)twoBytesToInt(fileData, f + 22);

    int numBytesPerSample = bitDepth / 8;

    // check that the audio format is PCM or Float
    if (audioFormat != WavAudioFormat::PCM && audioFormat !=
WavAudioFormat::IEEEFloat)
    {
        reportError("Error: This .WAV file is encoded in a format that this library does not
support!\n");
        return false;
    }

    // check the number of channels is mono or stereo
    if (numChannels < 1 || numChannels > 128)
    {
        reportError("Error: This WAV file seems to be an invalid number of channels (or
corrupted?).\n");
        return false;
    }

    // check header data is consistent
    if ((numBytesPerSecond != (numChannels * sampleRate * bitDepth) / 8) ||
(numBytesPerBlock != (numChannels * numBytesPerSample)))
    {
        reportError("Error: The header data in this WAV file seems to be inconsistent!\n");
        return false;
    }

    // check bit depth is either 8, 16, 24 or 32 bit
    if (bitDepth != 8 && bitDepth != 16 && bitDepth != 24 && bitDepth != 32)
    {
        reportError("Error: This file has a bit depth that is not 8, 16, 24 or 32 bits!\n");
        return false;
    }
```

```
    // ------------------------------------------------------
    // DATA CHUNK
    int d = indexOfDataChunk;
    std::string dataChunkID(fileData.begin() + d, fileData.begin() + d + 4);
    int32_t dataChunkSize = fourBytesToInt(fileData, d + 4);

    int numSamples = dataChunkSize / (numChannels * bitDepth / 8);
    int samplesStartIndex = indexOfDataChunk + 8;

    clearAudioBuffer();
    samples.resize(numChannels);

    for (int i = 0; i < numSamples; i++)
    {
        for (int channel = 0; channel < numChannels; channel++)
        {
            int sampleIndex = samplesStartIndex + (numBytesPerBlock * i) + channel *
numBytesPerSample;

            if (bitDepth == 8)
            {
                T sample = singleByteToSample(fileData[sampleIndex]);
                samples[channel].push_back(sample);
            }
            else if (bitDepth == 16)
            {
                int16_t sampleAsInt = twoBytesToInt(fileData, sampleIndex);
                T sample = sixteenBitIntToSample(sampleAsInt);
                samples[channel].push_back(sample);
            }
            else if (bitDepth == 24)
            {
                int32_t sampleAsInt = 0;
                sampleAsInt = (fileData[sampleIndex + 2] << 16) | (fileData[sampleIndex + 1]
<< 8) | fileData[sampleIndex];

                if (sampleAsInt & 0x800000) //  if the 24th bit is set, this is a negative number in
24-bit world
                    sampleAsInt = sampleAsInt | ~0xFFFFFF; // so make sure sign is extended to
the 32 bit float

                T sample = (T)sampleAsInt / (T)8388608.;
                samples[channel].push_back(sample);
            }
            else if (bitDepth == 32)
            {
                int32_t sampleAsInt = fourBytesToInt(fileData, sampleIndex);
```

```
            T sample;

            if (audioFormat == WavAudioFormat::IEEEFloat)
                sample = (T)reinterpret_cast<float&> (sampleAsInt);
            else // assume PCM
                sample = (T)sampleAsInt / static_cast<float>
(std::numeric_limits<std::int32_t>::max());

            samples[channel].push_back(sample);
        }
        else
        {
            assert(false);
        }
    }
  }

  // ----------------------------------------------------------
  // iXML CHUNK
  if (indexOfXMLChunk != -1)
  {
    int32_t chunkSize = fourBytesToInt(fileData, indexOfXMLChunk + 4);
    iXMLChunk = std::string((const char*)&fileData[indexOfXMLChunk + 8],
chunkSize);
  }

  return true;
}

//=============================================================
template <class T>
bool AudioFile<T>::decodeAiffFile(std::vector<uint8_t>& fileData)
{
  // ----------------------------------------------------------
  // HEADER CHUNK
  std::string headerChunkID(fileData.begin(), fileData.begin() + 4);
  //int32_t fileSizeInBytes = fourBytesToInt (fileData, 4, Endianness::BigEndian) + 8;
  std::string format(fileData.begin() + 8, fileData.begin() + 12);

  int audioFormat = format == "AIFF" ? AIFFAudioFormat::Uncompressed : format ==
"AIFC" ? AIFFAudioFormat::Compressed : AIFFAudioFormat::Error;

  // ----------------------------------------------------------
  // try and find the start points of key chunks
  int indexOfCommChunk = getIndexOfChunk(fileData, "COMM", 12,
Endianness::BigEndian);
  int indexOfSoundDataChunk = getIndexOfChunk(fileData, "SSND", 12,
Endianness::BigEndian);
```

```
    int indexOfXMLChunk = getIndexOfChunk(fileData, "iXML", 12,
Endianness::BigEndian);

    // if we can't find the data or format chunks, or the IDs/formats don't seem to be as
expected
    // then it is unlikely we'll able to read this file, so abort
    if (indexOfSoundDataChunk == -1 || indexOfCommChunk == -1 || headerChunkID !=
"FORM" || audioFormat == AIFFAudioFormat::Error)
    {
        reportError("Error: this doesn't seem to be a valid AIFF file");
        return false;
    }

    // ----------------------------------------------------------
    // COMM CHUNK
    int p = indexOfCommChunk;
    std::string commChunkID(fileData.begin() + p, fileData.begin() + p + 4);
    //int32_t commChunkSize = fourBytesToInt (fileData, p + 4, Endianness::BigEndian);
    int16_t numChannels = twoBytesToInt(fileData, p + 8, Endianness::BigEndian);
    int32_t numSamplesPerChannel = fourBytesToInt(fileData, p + 10,
Endianness::BigEndian);
    bitDepth = (int)twoBytesToInt(fileData, p + 14, Endianness::BigEndian);
    sampleRate = getAiffSampleRate(fileData, p + 16);

    // check the sample rate was properly decoded
    if (sampleRate == 0)
    {
        reportError("Error: This AIFF file has an unsupported sample rate!\n");
        return false;
    }

    // check the number of channels is mono or stereo
    if (numChannels < 1 || numChannels > 2)
    {
        reportError("Error: This AIFF file seems to be neither mono nor stereo (perhaps
multi-track, or corrupted?)!\n");
        return false;
    }

    // check bit depth is either 8, 16, 24 or 32-bit
    if (bitDepth != 8 && bitDepth != 16 && bitDepth != 24 && bitDepth != 32)
    {
        reportError("Error: This file has a bit depth that is not 8, 16, 24 or 32 bits!\n");
        return false;
    }

    // ----------------------------------------------------------
    // SSND CHUNK
```

```
    int s = indexOfSoundDataChunk;
    std::string soundDataChunkID(fileData.begin() + s, fileData.begin() + s + 4);
    int32_t soundDataChunkSize = fourBytesToInt(fileData, s + 4, Endianness::BigEndian);
    int32_t offset = fourBytesToInt(fileData, s + 8, Endianness::BigEndian);
    //int32_t blockSize = fourBytesToInt (fileData, s + 12, Endianness::BigEndian);

    int numBytesPerSample = bitDepth / 8;
    int numBytesPerFrame = numBytesPerSample * numChannels;
    int totalNumAudioSampleBytes = numSamplesPerChannel * numBytesPerFrame;
    int samplesStartIndex = s + 16 + (int)offset;

    // sanity check the data
    if ((soundDataChunkSize - 8) != totalNumAudioSampleBytes ||
totalNumAudioSampleBytes > static_cast<long>(fileData.size() - samplesStartIndex))
    {
        reportError("Error: The meta-data for this file doesn't seem right!\n");
        return false;
    }

    clearAudioBuffer();
    samples.resize(numChannels);

    for (int i = 0; i < numSamplesPerChannel; i++)
    {
        for (int channel = 0; channel < numChannels; channel++)
        {
            int sampleIndex = samplesStartIndex + (numBytesPerFrame * i) + channel *
numBytesPerSample;

            if (bitDepth == 8)
            {
                int8_t sampleAsSigned8Bit = (int8_t)fileData[sampleIndex];
                T sample = (T)sampleAsSigned8Bit / (T)128.;
                samples[channel].push_back(sample);
            }
            else if (bitDepth == 16)
            {
                int16_t sampleAsInt = twoBytesToInt(fileData, sampleIndex,
Endianness::BigEndian);
                T sample = sixteenBitIntToSample(sampleAsInt);
                samples[channel].push_back(sample);
            }
            else if (bitDepth == 24)
            {
                int32_t sampleAsInt = 0;
                sampleAsInt = (fileData[sampleIndex] << 16) | (fileData[sampleIndex + 1] << 8)
| fileData[sampleIndex + 2];
```

```
            if (sampleAsInt & 0x800000) //  if the 24th bit is set, this is a negative number in 24-bit world
                sampleAsInt = sampleAsInt | ~0xFFFFFF; // so make sure sign is extended to the 32 bit float

            T sample = (T)sampleAsInt / (T)8388608.;
            samples[channel].push_back(sample);
         }
         else if (bitDepth == 32)
         {
            int32_t sampleAsInt = fourBytesToInt(fileData, sampleIndex, Endianness::BigEndian);
            T sample;

            if (audioFormat == AIFFAudioFormat::Compressed)
               sample = (T)reinterpret_cast<float&> (sampleAsInt);
            else // assume uncompressed
               sample = (T)sampleAsInt / static_cast<float> (std::numeric_limits<std::int32_t>::max());

            samples[channel].push_back(sample);
         }
         else
         {
            assert(false);
         }
      }
   }

   // ----------------------------------------------------------
   // iXML CHUNK
   if (indexOfXMLChunk != -1)
   {
      int32_t chunkSize = fourBytesToInt(fileData, indexOfXMLChunk + 4);
      iXMLChunk = std::string((const char*)&fileData[indexOfXMLChunk + 8], chunkSize);
   }

   return true;
}

//=============================================================================
template <class T>
uint32_t AudioFile<T>::getAiffSampleRate(std::vector<uint8_t>& fileData, int sampleRateStartIndex)
{
   for (auto it : aiffSampleRateTable)
   {
```

```
        if (tenByteMatch(fileData, sampleRateStartIndex, it.second, 0))
            return it.first;
    }

    return 0;
}

//=============================================================
template <class T>
bool AudioFile<T>::tenByteMatch(std::vector<uint8_t>& v1, int startIndex1,
std::vector<uint8_t>& v2, int startIndex2)
{
    for (int i = 0; i < 10; i++)
    {
        if (v1[startIndex1 + i] != v2[startIndex2 + i])
            return false;
    }

    return true;
}

//=============================================================
template <class T>
void AudioFile<T>::clearAudioBuffer()
{
    for (size_t i = 0; i < samples.size(); i++)
    {
        samples[i].clear();
    }

    samples.clear();
}

//=============================================================
template <class T>
AudioFileFormat AudioFile<T>::determineAudioFileFormat(std::vector<uint8_t>&
fileData)
{
    std::string header(fileData.begin(), fileData.begin() + 4);

    if (header == "RIFF")
        return AudioFileFormat::Wave;
    else if (header == "FORM")
        return AudioFileFormat::Aiff;
    else
        return AudioFileFormat::Error;
}
```

```
//=============================================================
template <class T>
int32_t AudioFile<T>::fourBytesToInt(std::vector<uint8_t>& source, int startIndex,
Endianness endianness)
{
    int32_t result;

    if (endianness == Endianness::LittleEndian)
        result = (source[startIndex + 3] << 24) | (source[startIndex + 2] << 16) |
(source[startIndex + 1] << 8) | source[startIndex];
    else
        result = (source[startIndex] << 24) | (source[startIndex + 1] << 16) | (source[startIndex
+ 2] << 8) | source[startIndex + 3];

    return result;
}

//=============================================================
template <class T>
int16_t AudioFile<T>::twoBytesToInt(std::vector<uint8_t>& source, int startIndex,
Endianness endianness)
{
    int16_t result;

    if (endianness == Endianness::LittleEndian)
        result = (source[startIndex + 1] << 8) | source[startIndex];
    else
        result = (source[startIndex] << 8) | source[startIndex + 1];

    return result;
}

//=============================================================
template <class T>
int AudioFile<T>::getIndexOfString(std::vector<uint8_t>& source, std::string
stringToSearchFor)
{
    int index = -1;
    int stringLength = (int)stringToSearchFor.length();

    for (size_t i = 0; i < source.size() - stringLength; i++)
    {
        std::string section(source.begin() + i, source.begin() + i + stringLength);

        if (section == stringToSearchFor)
        {
            index = static_cast<int> (i);
            break;
```

```
        }
    }

    return index;
}

//=============================================================
template <class T>
int AudioFile<T>::getIndexOfChunk(std::vector<uint8_t>& source, const std::string&
chunkHeaderID, int startIndex, Endianness endianness)
{
    constexpr int dataLen = 4;
    if (chunkHeaderID.size() != dataLen)
    {
        assert(false && "Invalid chunk header ID string");
        return -1;
    }

    int i = startIndex;
    while (i < source.size() - dataLen)
    {
        if (memcmp(&source[i], chunkHeaderID.data(), dataLen) == 0)
        {
            return i;
        }

        i += dataLen;
        auto chunkSize = fourBytesToInt(source, i, endianness);
        i += (dataLen + chunkSize);
    }

    return -1;
}

//=============================================================
template <class T>
T AudioFile<T>::sixteenBitIntToSample(int16_t sample)
{
    return static_cast<T> (sample) / static_cast<T> (32768.);
}

//=============================================================
template <class T>
int16_t AudioFile<T>::sampleToSixteenBitInt(T sample)
{
    sample = clamp(sample, -1., 1.);
    return static_cast<int16_t> (sample * 32767.);
}
```

```
//=============================================================
template <class T>
uint8_t AudioFile<T>::sampleToSingleByte(T sample)
{
    sample = clamp(sample, -1., 1.);
    sample = (sample + 1.) / 2.;
    return static_cast<uint8_t> (sample * 255.);
}

//=============================================================
template <class T>
T AudioFile<T>::singleByteToSample(uint8_t sample)
{
    return static_cast<T> (sample - 128) / static_cast<T> (128.);
}

//=============================================================
template <class T>
T AudioFile<T>::clamp(T value, T minValue, T maxValue)
{
    value = std::min(value, maxValue);
    value = std::max(value, minValue);
    return value;
}

//=============================================================
template <class T>
void AudioFile<T>::reportError(std::string errorMessage)
{
    if (logErrorsToConsole) {
        std::cout << errorMessage << std::endl;
    }
}

template <class T>
void AudioFile<T>::ConcactChannels() {
    if (AudioFile<T>::isStereo()) {
        for (int sample_index = 0; sample_index < samples[1].size(); sample_index++) {
            samples[0].push_back(samples[1][sample_index]);
        }
        if (samples[0].size() != samples[1].size() * 2) {
            reportError("Error: Concacting channels failed!\n");
        }
        else {
            samples.erase(samples.begin() + 1);
        }
    }
```

```
}

#if defined (_MSC_VER)
__pragma(warning(pop))
#elif defined (__GNUC__)
_Pragma("GCC diagnostic pop")
#endif

#endif /* AudioFile_h */
```

Multifunction Audio Classifier

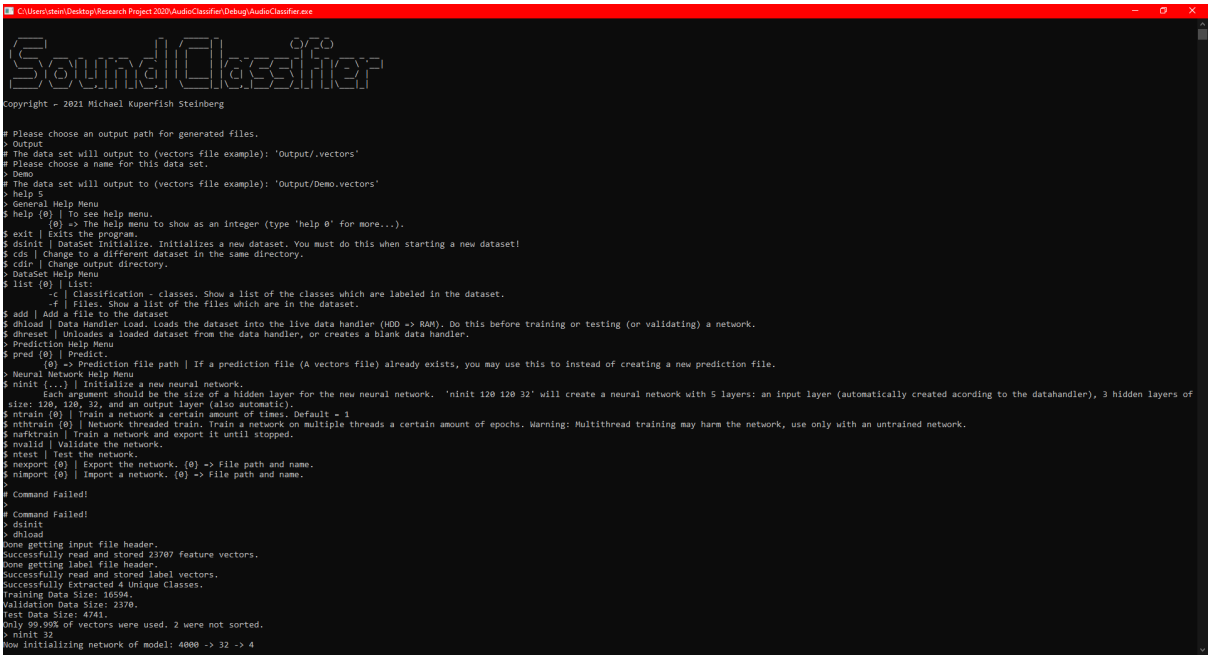
After some tinkering with the Music Genre Classifier, I realized there was more potential than I had anticipated; the exact same code and structure could be used to classify any type of sound! The entire project was written to be as generic and modular as possible, and therefore was easily adaptable to a few more problems.

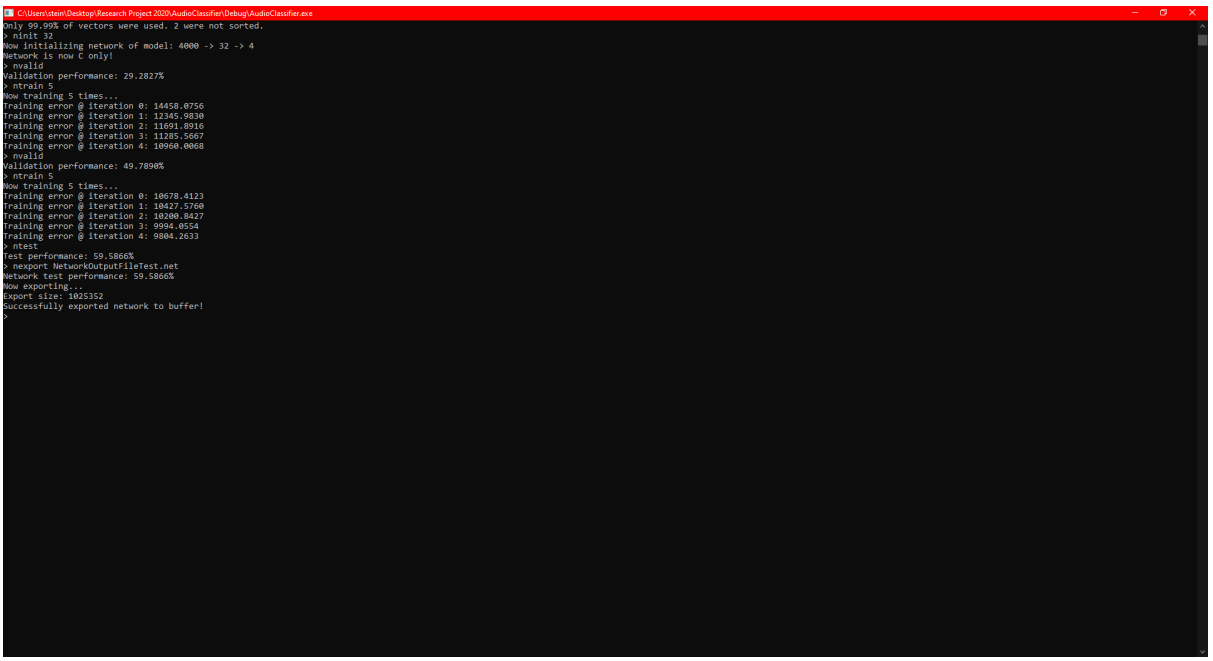
Network Output Files - .net

The only real difference between the various solutions to the problems is the resulting network file (.net file, explained below), which is produced uniquely for each problem and its dataset. This network file actually turned out to be invaluable as during testing, some versions of the network required multiple days of training. Unfortunately, the power lines in Tel Aviv are extremely volatile during winter (around when I began training the networks), and the power would shut off every few hours. This meant I would lose hours of work each time the computer crashed. To overcome this, the network export files were designed. Network export files contain all the information needed to recreate a “carbon” copy of a trained network.

A Network Output file contains the weights and biases of each and every neuron, and information needed to redesign the structure (layer layout) of the network. Additionally, to improve the ability to sort between network output files, before each export the network tests itself and appends its best score into the output file.

Final Result:





Using this Audio Classifier, we are able to classify:

- Different genres of music:
  - The first test was to see if we can differentiate between genres of music. The result was that we can classify with a little over 78% accuracy ‘Classical Music’, ‘Electronic Dance Music’, ‘Jazz’, and ‘Rock’. This test was done with a network of 4000 -> 420 -> 32 -> 4, and took around 11 training iterations on a dataset of 23,707 feature vectors.
    - Network File: MNIST_Network_Output3.net
- Different people speaking:
  - The next test was to differentiate between people’s voices. As explained in the Theoretical Background, this is almost identical to our previous problem. Hence, we used the exact same program, with the exact same network configuration to give us an accuracy of over 96% when differentiating between ‘Angela Merkel’, ‘Martin Luther King Jr’, ‘John F Kennedy’, and ‘Bucky Roberts’, after around 30 training cycles of 2,838 data samples.
    - Network File: FourV_96.net
- Different instruments:
  - Now let’s try instruments! Testing with these following classifications and files Piano, Guitar, Violin, returned a test accuracy of ~98% on a very simple network of 4000 -> 30 -> 3 with 7547 feature vectors.
    - Network file: Instr98.net

Code:

neuron.h + neuron.cpp

```
#pragma once

#include <stdio.h>
#include <vector>
#include <cmath>
#include <random>
#include <time.h>

static int import_c_only = 1;

class Neuron
{
public:
    double output;
    double delta;
    std::vector<double> weights;
    double * weights_array;
    uint32_t weights_array_size;

    Neuron(int, int); // ctor
    Neuron(); // ctor for import

    void initialize_weights(int); // randomize weight values

    double activate(std::vector<double> inputs);
    double activate(int inputs_size, double * inputs); // c

    void c_only();


    uint32_t export_neuron(char * buffer);
    void import_neuron(uint32_t prev_layer_size, uint32_t layer_size, char * buffer);
    uint64_t raw_size();
};
```

```
#include "neuron.h"

double generateRandomNumber(double min, double max)
{
    double random = (double)rand() / RAND_MAX;
    return min + random * (max - min);
```

```
}

Neuron::Neuron() {}

Neuron::Neuron(int prev_layer_size, int curr_layer_size)
{
    initialize_weights(prev_layer_size);
}
void Neuron::initialize_weights(int prev_layer_size)
{
    for (int i = 0; i < prev_layer_size + 1; i++)
    {
        weights.push_back(generateRandomNumber(-1.0, 1.0));
    }
}

double Neuron::activate(std::vector<double> inputs) {
    double activation = weights.back(); // bias
    for (int i = 0; i < weights.size() - 1; i++) {
        activation += weights[i] * inputs[i];
    }
    return activation;
}
double Neuron::activate(int input_size, double * inputs) {
    double activation = weights_array[input_size]; // bias
    for (int i = 0; i < input_size; i++) {
        activation += weights_array[i] * inputs[i];
    }
    return activation;
}

void Neuron::c_only() {
    if (weights.size() > 0) {
        weights_array = (double*)malloc(weights.size() * sizeof(double));
        weights_array_size = weights.size();
        for (int i = 0; i < weights.size(); i++) {
            weights_array[i] = weights.at(i);
        }
        weights.clear();
    }
}

uint64_t Neuron::raw_size() {
    return (weights_array_size * sizeof(weights_array[0]));
}

uint32_t Neuron::export_neuron(char * buffer) {
    memcpy(buffer, weights_array, (weights_array_size * sizeof(weights_array[0])));
```

```
    return (weights_array_size * sizeof(weights_array[0]));
}

void Neuron::import_neuron(uint32_t prev_layer_size, uint32_t layer_size, char * buffer) {
    if (weights_array != NULL) { free(weights_array); }
    weights_array_size = prev_layer_size + 1;
    weights_array = (double*)malloc(weights_array_size * sizeof(double));

    for (int i = 0; i < weights_array_size; i++) {
        double val = 0;
        memcpy(&val, buffer + (i * sizeof(val)), sizeof(val));
        if (import_c_only) {
            weights_array[i] = val;
        } else { weights.push_back(val); }

    }
}
```

layer.h + layer.cpp

```
#pragma once

#include "neuron.h"
#include <stdint.h>
#include <vector>

static int layerId = 0;

class Layer
{
public:
    int id;

    int current_layer_size;
    std::vector<Neuron *> neurons;
    std::vector<double> layer_outputs;

    Neuron ** neurons_array;
    int neurons_size;
    double * layer_outputs_array;

    Layer(int prev_layer_size, int curr_layer_size); // ctor
    Layer(int, int, bool); // ctor for import
```

```
    void c_only();
};
```

```
#include "layer.h"

Layer::Layer(int prev_layer_size, int curr_layer_size)
{
    for (int i = 0; i < curr_layer_size; i++)
    {
        neurons.push_back(new Neuron(prev_layer_size, curr_layer_size));
    }

    this->current_layer_size = curr_layer_size;
}

Layer::Layer(int prev_layer_size, int curr_layer_size, bool importing)
{
    if (importing) {
        if (import_c_only) {
            if (neurons_array != NULL) { free(neurons_array); }
            neurons_size = curr_layer_size;
            neurons_array = (Neuron**)malloc(neurons_size *sizeof(Neuron*));
        }

        for (int i = 0; i < curr_layer_size; i++)
        {
            if (import_c_only) {
                neurons_array[i] = new Neuron();
            }
            else {
                neurons.push_back(new Neuron());
            }
        }
    }
    else {
        for (int i = 0; i < curr_layer_size; i++)
        {
            neurons.push_back(new Neuron(prev_layer_size, curr_layer_size));
        }
    }
    this->current_layer_size = curr_layer_size;
    this->id = ++layerId;
}

void Layer::c_only() {
    neurons_size = neurons.size();

    neurons_array = (Neuron**)malloc(neurons_size * sizeof(Neuron*));
```

```
    for (int i = 0; i < neurons_size; i++) {
        neurons_array[i] = neurons.at(i);
        neurons_array[i]->c_only();
    }

    layer_outputs_array = (double *)malloc(neurons_size * sizeof(double));

    neurons.clear();
    layer_outputs.clear();
}
```

network.h + network.cpp

```
#pragma once

#include "Data/data.h"
#include "Data/data_handler.h"
#include "Data/common_data.h"
#include "neuron.h"
#include "layer.h"

#include <numeric>
#include <algorithm>
#include <thread>

class Network : public Common_data
{
public:
    std::vector<Layer *> layers;
    Layer ** layers_array;
    int layers_size;
    double learning_rate;
    double test_performance;

    Network(double); // ctor for import
    Network(std::vector<int> hidden_layers_specification, int, int, double); // ctor
    ~Network(); // dtor

    void c_only();

    std::vector<double> fprop(Data * d); // forward propagation
    void bprop(std::vector<double> deriv_errors); // back propagation
    void update_weights(Data * data); // update weights after bprop
```

```
    int fprop_c(Data * d, double ** output); // forward propagation - c | returns class count
    void bprop_c(double * deriv_errors); // back propagation - c
    void update_weights_c(Data * data); // update weights after bprop - c

    double train(); // returns error
    double validate(); // return percentage correct
    double test(); // return percentage correct

    double train_c(); // returns error - c
    double train_c(int target); // Train only datasamples with this label | returns error - c
    double validate_c(); // return percentage correct - c
    double test_c(); // return percentage correct - c

    double transfer_activation(double activat);

    int predict(Data * data);
    int predict_c(Data * data);

    int real_predict();

    int export_network(char ** buffer); // returns the size of buffer
    void import_network(char * buffer);
};
```

```
#include "network.h"

Network::~Network() {}
Network::Network(std::vector<int> hidden_layers_specification, int input_size, int
number_of_classes, double learning_rate) {
    for (int i = 0; i < hidden_layers_specification.size(); i++)
    {
        if (i == 0)
            layers.push_back(new Layer(input_size, hidden_layers_specification.at(i)));
        else
            layers.push_back(new Layer(layers.at(i - 1)->neurons.size(),
hidden_layers_specification.at(i)));

    }
    layers.push_back(new Layer(layers.at(layers.size() - 1)->neurons.size(),
number_of_classes));

    this->learning_rate = learning_rate;
}
Network::Network(double learning_rate) {
    this->learning_rate = learning_rate;
}
std::vector<double> Network::fprop(Data * d) {
```

```
    std::vector<double> inputs = *d->get_normalized_feature_vector();

    for (int i = 0; i < layers.size(); i++)
    {
        Layer * l = layers.at(i);
        std::vector<double> new_inputs;
        int neuron_index = 0;
        for (Neuron * n : l->neurons)
        {
            double activation = n->activate(inputs);
            n->output = this->transfer_activation(activation);
            new_inputs.push_back(n->output);
        }
        //l->layer_outputs = new_inputs;
        inputs = new_inputs;
    }
    return inputs; // output layer outputs
}
int Network::fprop_c(Data * d, double ** output) { // fprop with only c
    int inputs_size = d->get_feature_array_size();
    double * inputs = (double*)malloc(inputs_size * sizeof(double));

    memcpy(inputs, d->get_normalized_feature_array(), inputs_size * sizeof(double));

    for (int layer_index = 0; layer_index < layers_size; layer_index++) {
        Layer * l = (layers_array[layer_index]);
        double * new_inputs = (double *)malloc(l->neurons_size * sizeof(double));
        for (int neuron_index = 0; neuron_index < l->neurons_size; neuron_index++) {
            Neuron * n = (l->neurons_array[neuron_index]);
            double activation = n->activate(inputs_size, inputs);
            n->output = this->transfer_activation(activation);
            new_inputs[neuron_index] = n->output;
        }
        inputs_size = l->neurons_size;
        //memcpy(l->layer_outputs_array, new_inputs, inputs_size * sizeof(double));
        inputs = (double *)realloc(inputs, inputs_size * sizeof(double));
        memcpy(inputs, new_inputs, inputs_size * sizeof(double));
        free(new_inputs);
    }
    *output = (double *)malloc(inputs_size * sizeof(double));
    memcpy(*output, inputs, inputs_size * sizeof(double));
    free(inputs);

    return inputs_size;
}

void Network::bprop(std::vector<double> deriv_errors) {
    for (int i = layers.size() - 1; i >= 0; i--)
```

```
    {
        Layer * l = layers.at(i);
        std::vector<double> errors;
        if (i != layers.size() - 1)
        {
            for (int j = 0; j < l->neurons.size(); j++)
            {
                double error = 0.0;
                for (Neuron *n : layers.at(i + 1)->neurons)
                {
                    error += (n->weights.at(j) * n->delta);
                }
                errors.push_back(error);
            }
        }
        else {
            // If is last layer (i == layers.size()-1)
            errors = deriv_errors;
        }

        for (int j = 0; j < l->neurons.size(); j++)
        {
            Neuron * n = l->neurons.at(j);
            n->delta = errors[j];
        }
    }
}
void Network::bprop_c(double * deriv_errors) {
    double * errors;
    for (int i = layers_size - 1; i >= 0; i--)
    {
        Layer * l = (layers_array[i]);

        if (i != layers_size - 1)
        {
            errors = (double *)malloc(l->neurons_size * sizeof(double));
            for (int j = 0; j < l->neurons_size; j++)
            {
                /// for each neuron in current layer.

                double error = 0.0;
                for (int k = 0; k < layers_array[i + 1]->neurons_size; k++)
                {
                    /// for each neuron in next layer
                    Neuron * n = (layers_array[i + 1]->neurons_array[k]);
                    error += (n->weights_array[j] * n->delta);
                }
                errors[j] = error;
```

```
            }
        }
        else {
            // If is last layer (i == layers.size()-1)
            errors = deriv_errors;
        }

        for (int j = 0; j < l->neurons_size; j++)
        {
            Neuron * n = (l->neurons_array[j]);
            n->delta = errors[j];
        }

        if (deriv_errors != errors) {
            free(errors);
        }
    }
}
void Network::update_weights(Data * d) {
    std::vector<double> inputs = *d->get_normalized_feature_vector();
    for (int i = 0; i < layers.size(); i++)
    {
        if (i != 0)
        {
            for (Neuron *n : layers.at(i - 1)->neurons)
            {
                inputs.push_back(n->output);
            }
        }
        for (Neuron *n : layers.at(i)->neurons)
        {
            for (int j = 0; j < inputs.size(); j++)
            {
                n->weights.at(j) += this->learning_rate * n->delta * inputs.at(j);
            }
            n->weights.back() += this->learning_rate * n->delta;
        }
        inputs.clear();
    }
}
void Network::update_weights_c(Data * d) {
    int inputs_size = d->get_feature_array_size();
    double * inputs = (double *)malloc(inputs_size * sizeof(double));
    memcpy(inputs, d->get_normalized_feature_array(), inputs_size * sizeof(double));

    for (int i = 0; i < layers_size; i++)
    {
        if (i != 0)
```

```
        {
            inputs_size = layers_array[i - 1]->neurons_size;
            inputs = (double *)realloc(inputs, inputs_size * sizeof(double));

            for (int j = 0; j < layers_array[i - 1]->neurons_size; j++)
            {
                Neuron * n = (layers_array[i - 1]->neurons_array[j]);
                inputs[i] = n->output;
            }
        }
        for (int j = 0; j < layers_array[i]->neurons_size; j++)
        {
            Neuron * n = (layers_array[i]->neurons_array[j]);
            for (int k = 0; k < inputs_size; k++)
            {
                n->weights_array[k] += this->learning_rate * n->delta * inputs[k];
            }
            n->weights_array[inputs_size] += this->learning_rate * n->delta;
        }
    }

    free(inputs);
}

double Network::train() {
    double sum_error = 0.0;
    std::vector<double> deriv_error;

    int output_vector_size = this->training_data->at(0)->get_class_vector()->size();
    for (int i = 0; i < output_vector_size; i++) { deriv_error.push_back(0.0); }

    for (Data * d : *this->training_data)
    {
        std::vector<double> outputs = fprop(d);
        std::vector<int> expected = *(d->get_class_vector());
        double tmp_sum_error = 0.0;
        for (int j = 0; j < outputs.size(); j++)
        {
            tmp_sum_error += pow((double)expected.at(j) - outputs.at(j), 2);
            deriv_error[j] += (double)expected.at(j) - outputs.at(j);
        }
        sum_error += tmp_sum_error;

        for (int j = 0; j < output_vector_size; j++)
        {
            deriv_error.at(j) /= this->training_data->size();
            deriv_error.at(j) *= 2;
        }
```

```
        bprop(deriv_error);
        update_weights(d);

        for (int j = 0; j < output_vector_size; j++)
        {
            deriv_error.at(j) = 0;
        }
    }

    return sum_error;
}
double Network::validate() {
    double num_correct = 0.0;
    double count = 0.0;
    for (Data * d : *this->validation_data)
    {
        count++;
        int index = predict(d);
        if (d->get_class_vector()->at(index) == 1) { num_correct++; }
    }

    return num_correct / count;
}
double Network::test() {
    double num_correct = 0.0;
    double count = 0.0;
    for (Data * d : *this->test_data)
    {
        count++;
        int index = predict(d);
        if (d->get_class_vector()->at(index) == 1) { num_correct++; }
    }

    return num_correct / count;
}

double Network::train_c() {
    double sum_error = 0.0;
    int output_vector_size = this->training_data->at(0)->get_class_array_size();
    double * deriv_error = (double *)calloc(output_vector_size, sizeof(double));

    for (Data * d : *this->training_data)
    {
        double * outputs;
        fprop_c(d, &outputs);

        int * expected = d->get_class_array();
```

```
        double tmp_sum_error = 0.0;
        for (int j = 0; j < output_vector_size; j++)
        {
            double exp = (double)expected[j];
            double out = outputs[j];
            tmp_sum_error += pow(exp - out, 2);
            deriv_error[j] += exp - out;
        }
        sum_error += tmp_sum_error;

        for (int j = 0; j < output_vector_size; j++)
        {
            deriv_error[j] /= this->training_data->size();
            deriv_error[j] *= 2;
        }

        bprop_c(deriv_error);
        update_weights_c(d);

        for (int j = 0; j < output_vector_size; j++)
        {
            deriv_error[j] = 0;
        }

        free(outputs);
    }

    free(deriv_error);

    return sum_error;
}

double Network::train_c(int target) {
    double sum_error = 0.0;
    int output_vector_size = this->training_data->at(0)->get_class_array_size();
    double * deriv_error = (double *)calloc(output_vector_size, sizeof(double));

    for (Data * d : *this->training_data)
    {
        if (d->get_label() != target) { continue; } // Skip anything which does not match the
target label


        double * outputs;
        fprop_c(d, &outputs);

        int * expected = d->get_class_array();
        double tmp_sum_error = 0.0;
```

```
        for (int j = 0; j < output_vector_size; j++)
        {
            double exp = (double)expected[j];
            double out = outputs[j];
            tmp_sum_error += pow(exp - out, 2);
            deriv_error[j] += exp - out;
        }
        sum_error += tmp_sum_error;

        for (int j = 0; j < output_vector_size; j++)
        {
            deriv_error[j] /= this->training_data->size();
            deriv_error[j] *= 2;
        }

        bprop_c(deriv_error);
        update_weights_c(d);

        for (int j = 0; j < output_vector_size; j++)
        {
            deriv_error[j] = 0;
        }

        free(outputs);
    }

    free(deriv_error);

    return sum_error;
}

double Network::validate_c() {
    double num_correct = 0.0;
    double count = 0.0;
    for (Data * d : *this->validation_data)
    {
        count++;
        int index = predict_c(d);
        if (d->get_class_array()[index] == 1) { num_correct++; }
    }

    return num_correct / count;
}
double Network::test_c() {
    double num_correct = 0.0;
    double count = 0.0;
    for (Data * d : *this->test_data)
    {
```

```
        count++;
        int index = predict_c(d);
        if (d->get_class_array()[index] == 1) { num_correct++; }
    }

    test_performance = (num_correct / count);

    return num_correct / count;
}

double Network::transfer_activation(double activat) {
    return 1.0 / (1.0 + exp(-activat));
}

int Network::predict(Data * data) {
    std::vector<double> outputs = fprop(data);
    return std::distance(outputs.begin(), std::max_element(outputs.begin(), outputs.end()));
}
int Network::predict_c(Data * data) {
    double * outputs;
    int class_count = fprop_c(data, &outputs);

    int max_ind = 0;
    double max = 0;
    for (int i = class_count - 1; i >= 0; i--) {
        if (outputs[i] > max) { max_ind = i; max = outputs[i]; }
    }

    free(outputs);

    return max_ind;
}

int Network::real_predict()
{
    // A map to predictions and the amout of times they were made
    std::map<int, double> predictions;

    double numCorrect = 0.0;
    double count = 0.0;
    int size = prediction_data->size();
    for (Data *data : *this->prediction_data)
    {
        Data_Handler::print_loading(false, count, size);
        count++;
        double * outputs;
        int outputs_size = fprop_c(data, &outputs);
```

```
        for (int i = 0; i < outputs_size; i++) {
            if (count == 0) {
                predictions[i] = outputs[i];
            }
            else {
                predictions[i] += outputs[i];
            }
        }
    }

    printf("\r                \r");

    double total = 0;
    for (auto tpl : predictions) {
        total += tpl.second;
    }

    int top_guessed = predictions.begin()->first;
    double top_guessed_count = predictions.begin()->second;

    printf("Prediction table: \n");
    for (auto tpl : predictions) {
        printf("\t%u) %.2f%%\n", tpl.first, (float)(tpl.second * 100.0 / total));
        if (tpl.second > top_guessed_count) {
            top_guessed = tpl.first;
            top_guessed_count = tpl.second;
        }
    }

    return top_guessed;
}

void Network::c_only() {
    layers_size = layers.size();
    layers_array = (Layer **)malloc(layers_size * sizeof(Layer *));
    for (int i = 0; i < layers_size; i++) {
        layers_array[i] = (layers.at(i));
    }

    layers.clear();

    for (int i = 0; i < layers_size; i++) {
        Layer * l = layers_array[i];
        l->c_only();
    }

    printf("Network is now C only!\n");
}
```

```
int Network::export_network(char ** buffer) {
    printf("Now exporting...\n");
    unsigned long long int total_size = 0;

    total_size += sizeof(total_size);

    total_size += sizeof(double); // network_accuracy

    uint32_t layer_size = 0;
    uint32_t prev_layer_size = layers_array[0]->neurons_array[0]->weights_array_size-1;

    total_size += sizeof(layers_size);

    total_size += sizeof(prev_layer_size);

    for (int i = 0; i < layers_size; i++)
    {
        layer_size = layers_array[i]->current_layer_size;

        total_size += 1 * sizeof(layer_size); // layer_size

        for (int j = 0; j < layers_array[i]->neurons_size; j++) {
            Neuron * n = layers_array[i]->neurons_array[j];
            total_size += n->raw_size();
        }

        prev_layer_size = layer_size;

        total_size += 1 * sizeof(prev_layer_size); // prev_layer_size
    }

    printf("Export size: %llu\n", total_size);

    *buffer = (char *)calloc(total_size, sizeof(char));



    uint32_t current_size_index = 0;

    memcpy(*buffer, &total_size, sizeof(total_size));
    current_size_index += sizeof(total_size);

    double network_accuracy = this->test_performance;
    memcpy(*buffer + current_size_index, &network_accuracy, sizeof(network_accuracy));
    current_size_index += sizeof(network_accuracy);

    layer_size = 0;
```

```
    prev_layer_size = layers_array[0]->neurons_array[0]->weights_array_size - 1;

    uint32_t layer_sizes = layers_size;
    memcpy(*buffer + current_size_index, &layer_sizes, sizeof(layer_sizes));
    current_size_index += 1 * sizeof(layer_sizes); // layer_sizes

    memcpy(*buffer + current_size_index, &prev_layer_size, sizeof(prev_layer_size));
    current_size_index += 1 * sizeof(uint32_t); // prevlayerSize

    for (int i = 0; i < layers_size; i++)
    {
        layer_size = layers_array[i]->current_layer_size;

        memcpy(*buffer + current_size_index, &layer_size, sizeof(layer_size));
        current_size_index += 1 * sizeof(uint32_t); // layerSize

        for (int j = 0; j < layers_array[i]->neurons_size; j++) {
            Neuron * n = layers_array[i]->neurons_array[j];
            current_size_index += n->export_neuron((*buffer) + current_size_index);
        }

        prev_layer_size = layer_size;

        memcpy(*buffer + current_size_index, &prev_layer_size, sizeof(prev_layer_size));
        current_size_index += 1 * sizeof(uint32_t); // prevlayerSize
    }

    if (current_size_index != total_size) {
        printf("Total size does not match!\n");
    }
    else {
        /*FILE * f;
        errno_t err = fopen_s(&f, "Network_Output.net", "wb");
        fwrite(*buffer, sizeof(char), total_size, f);
        fclose(f);*/
        printf("Successfully exported network to buffer!\n");
        return total_size;
    }
}

void Network::import_network(char * buffer) {
    uint32_t current_size_index = 0;
    unsigned long long total_size = 0;
    memcpy(&total_size, buffer, sizeof(total_size));
    current_size_index += sizeof(total_size);

    double network_accuracy = 0;
    memcpy(&network_accuracy, buffer + current_size_index, sizeof(network_accuracy));
```

```
    current_size_index += sizeof(network_accuracy);

    uint32_t layer_sizes = 0;
    memcpy(&layer_sizes, buffer + current_size_index, sizeof(layer_sizes));
    current_size_index += sizeof(layer_sizes);

    uint32_t layer_size = 0;
    uint32_t prev_layer_size = 0;

    memcpy(&prev_layer_size, buffer + current_size_index, sizeof(prev_layer_size));
    current_size_index += 1 * sizeof(uint32_t); // prev_layer_size

    //inputLayer = new InputLayer(0, prev_layer_size, true);

    if (layers_array != NULL) { free(layers_array); }
    layers_size = layer_sizes; // netw->layers_size := layer_size
    layers_array = (Layer**)malloc(layers_size * sizeof(Layer*));

    for (int i = 0; i < layer_sizes; i++)
    {
        memcpy(&layer_size, buffer + current_size_index, sizeof(layer_size));
        current_size_index += 1 * sizeof(uint32_t); // layerSize

        Layer * layer = new Layer(prev_layer_size, layer_size, true);

        for (int j = 0; j < layer->neurons_size; j++) {
            Neuron * n = layer->neurons_array[j];
            n->import_neuron(prev_layer_size, layer_size, buffer + current_size_index);
            current_size_index += n->raw_size();
        }

        layers_array[i] = layer;

        memcpy(&prev_layer_size, buffer + current_size_index, sizeof(prev_layer_size));
        current_size_index += 1 * sizeof(uint32_t); // prev_layer_size
    }

    if (current_size_index != total_size) {
        printf("Import sizes do not match!\n");
    }
    else {
        printf("Successfully imported network with accuracy: %.4f!\n", network_accuracy);
    }
}
```

CommandLineFunctions.h + CommandLineFunctions.cpp

```
#pragma once

//#ifndef __CommandLineFunctions
//#define __CommandLineFunctions


#include <iostream>
#include <sys/stat.h>
#include <time.h>
#include <vector>
#include <map>

#include "AudioFile.h"
#include "FFT.h"

#include "Neural_Network/network.h"

/// Globals

// Reference to global neural network
static Network * network;

// Reference to global dataset
static Data_Handler * data_handler;


// Global strings (character arrays)
static char DataSetLabel[128] = { 0 };
static char OutputPath[128] = { 0 };
static char fn_label_map[128];
static char fn_data_map[128];
static char fn_vectors[128];
static char fn_labels[128];


/// Helping_Funcs
std::string get_current_dir();
inline bool file_exists(const std::string& name);
int getFileSize(FILE * f); // This will move the file pointer to SEEK_SET + 0
uint32_t convert_to_big_endian(uint32_t b);
uint32_t convert_to_little_endian(uint32_t b);
// Change the last character of a string into a null byte (if it's a newline).
void fix_fgets(char * inp);
```

```
// Check if an input matches a command
bool cmpcommand(char * inp, const char * command);

// Count the amount of arguments given in an input command (as ' ' delimiter)
int CountCommandArgs(char * command);

// Get command input argument at index as place it into output (also allocate memory for
output)
int GetCommandArg(char * command, int index, char ** output);

/// Command_Line_Commands

// Update file paths using the user specified outputpath and the dataset global name
int UpdateFilePaths();

// Change the path of the output directory
int ChangeOutputPath();

// Change the dataset's name
int ChangeDataSetName();

// Count how many labels exist in the label map file.
int getExistingLabelsCount(uint8_t * existing_labels, FILE * fLabelMap);

// Proccess 'list' commands
int ListView(char * command);

// Prints the classes and their labels, which are stored in the label map file.
int ViewClassifications();

// Prints the file paths of each file in the dataset sorted by their labels.
int ViewDataFiles();

// Add an item to a dataset
int AddDataToSet();

// Initialize a new dataset
int Initialize();

// predict the label of a file using the Classifier
/*
   The predict function can take arguments such as:
   1) a file path.
      The first input can be the relative path to a vectors file which you want to use instead
of processing a wav file into a vectors file.
   2) the '-f' argument.
      This is passed into the classifier as 'fast' which tells the program to only check some of
```

```
the data points (arbitrarily) and not all of them.
        This obviously results in faster execution time (10x faster - since we take only a tenth
of the vectors), but there is a penalty to the accuracy of the results. Though the final
prediction is still usually correct.
*/
int Predict(char * command);

// Initialize Network of size / model
int InitNetwork(char * command);

// Load an existing network
int LoadNetwork(char * command);

// Export the loaded network
int ExportNetwork(char * command);

// Train the loaded network
int TrainNetwork(char * command);

// Validate the loaded network
int ValidateNetwork();

// Train and Export the network
int AFKTrain();

// Load dataset into data_handler
int LoadDataHandler();

// Load predict file
int LoadPredictFile(char * command);

// Exit
int ExitProgram();

enum HelpMenus {
    NoHelpMenu,
    GeneralHelpMenu,
    DataSetHelpMenu,
    PreidctionHelpMenu,
    NetworkHelpMenu,
    AllHelpMenu
};

// Show help menu
int Help(HelpMenus helpMenu);
int Help(char * command);
```

```
// Gets user input and compares it to valid commands
int ProccessCommands();


//#endif // !__CommandLineFunctions
```

```
#include "CommandLineFunctions.h"

#include <direct.h>
#define GetCurrentDir _getcwd

#define MaximumLabelLength 255

std::string get_current_dir() {
   char buff[FILENAME_MAX];
   GetCurrentDir(buff, FILENAME_MAX);
   std::string current_working_dir(buff);
   return current_working_dir;
}
inline bool file_exists(const std::string& name) {
   struct stat buffer;
   return (stat(name.c_str(), &buffer) == 0);
}

int getFileSize(FILE * f) { // This will move the file pointer to SEEK_SET + 0
   fseek(f, 0, SEEK_END);
   int size = ftell(f);
   fseek(f, 0, SEEK_SET);
   return size;
}

uint32_t convert_to_big_endian(uint32_t b) {
   unsigned char bytes[4];
   memcpy(bytes, &b, sizeof(b));
   return (uint32_t)((bytes[0] << 24) | (bytes[1] << 16) | (bytes[2] << 8) | (bytes[3]));
}
uint32_t convert_to_little_endian(uint32_t b) {
   unsigned char bytes[4];
   memcpy(bytes, &b, sizeof(b));
   return (uint32_t)((bytes[0] << 24) | (bytes[1] << 16) | (bytes[2] << 8) | (bytes[3]));
}

// Change the last character of a string into a null byte (if it's a newline).
void fix_fgets(char * inp) {
   if (inp[strlen(inp) - 1] == '\n') { inp[strlen(inp) - 1] = 0x0; }
}

// Check if an input matches a command
```

```
bool cmpcommand(char * inp, const char * command) {
    if (memcmp(inp, command, strlen(command)) == 0) {
        char fin_char = inp[strlen(command)];
        // the format specified is that a command ends with one of the following four
characters.
        if (fin_char == ' ' || fin_char == '\t' || fin_char == '\n' || fin_char == '\0') {
            return true;
        }
    }
    return false;
}

// Count the amount of arguments given in an input command (as ' ' delimiter)
int CountCommandArgs(char * command) {
    char * tmpCommand = (char *)malloc((strlen(command) + 1) * sizeof(char));
    memcpy(tmpCommand, command, strlen(command) + 1);

    int count = 0;
    const char delim[2] = " "; // delimiter
    char * next_tok;
    char * tok = strtok_s(tmpCommand, delim, &next_tok);
    while (tok != NULL) {
        //printf("Arg #%d: '%s'\n", count, tok);
        count++;
        tok = strtok_s(NULL, delim, &next_tok);
    }

    free(tmpCommand);

    return count;
}

// Get command input argument at index as place it into output (also allocate memory for
output)
int GetCommandArg(char * command, int index, char ** output) {
    char * tmpCommand = (char *)malloc((strlen(command) + 1) * sizeof(char));
    memcpy(tmpCommand, command, strlen(command) + 1);

    int count = 0;
    const char delim[2] = " ";
    char * next_tok;
    char * tok = strtok_s(tmpCommand, delim, &next_tok);
    while (tok != NULL && count < index) {
        //printf("Arg #%d: '%s'\n", count, tok);
        count++;
        tok = strtok_s(NULL, delim, &next_tok);
    }
```

```
    if (index > count) { free(tmpCommand); return -1; }

    *output = (char*)malloc((strlen(tok) + 1) * sizeof(char));
    memcpy(*output, tok, strlen(tok) + 1);

    free(tmpCommand);
    return 0;
}

// Update file paths using the user specified outputpath and the dataset global name
int UpdateFilePaths() {
    /*fn_label_map = (char*)calloc(128, sizeof(char));
    fn_data_map = (char*)calloc(128, sizeof(char));
    fn_vectors = (char*)calloc(128, sizeof(char));
    fn_labels = (char*)calloc(128, sizeof(char));*/

    sprintf_s(fn_label_map, 128, "%s/%s.label_map", OutputPath, DataSetLabel);
    sprintf_s(fn_data_map, 128, "%s/%s.data_map", OutputPath, DataSetLabel);
    sprintf_s(fn_vectors, 128, "%s/%s.vectors", OutputPath, DataSetLabel);
    sprintf_s(fn_labels, 128, "%s/%s.labels", OutputPath, DataSetLabel);

    return 0;
}
int ChangeOutputPath() {
    printf("# Please choose an output path for generated files.\n> ");
    memset(OutputPath, 0x0, 128 * sizeof(char));
    fgets(OutputPath, 127, stdin);
    fix_fgets(OutputPath);
    UpdateFilePaths();
    printf("# The data set will output to (vectors file example): '%s'\n", fn_vectors);
    return 0;
}
int ChangeDataSetName() {
    printf("# Please choose a name for this data set.\n> ");
    memset(DataSetLabel, 0x0, 128 * sizeof(char));
    fgets(DataSetLabel, 127, stdin);
    fix_fgets(DataSetLabel);
    UpdateFilePaths();
    printf("# The data set will output to (vectors file example): '%s'\n", fn_vectors);
    return 0;
}

// Count how many labels exist in the label map file.
int getExistingLabelsCount(uint8_t * existing_labels, FILE * fLabelMap) {
    int existing_labels_count = 0;
    char label_pair[MaximumLabelLength + 2];
```

```
    while (fread_s(label_pair, (MaximumLabelLength + 2) * sizeof(char), sizeof(char),
MaximumLabelLength + 1, fLabelMap)) {
        uint8_t label_key = label_pair[0];
        existing_labels[existing_labels_count] = label_key;
        existing_labels_count++;
    }
    return existing_labels_count;
}

// Prints the classes and their labels, which are stored in the label map file.
int ViewClassifications() {
    printf("# The following classifications exist in the data set: \n\n");

    uint8_t existing_labels[256];
    uint8_t existing_labels_count = 0;

    // A file with a map (dictionary) of the label numbers (0-255) and their corresponding
"human" definition.
    FILE * fLabelMap;
    errno_t errLabelMap;
    // A file with a map of the actual wav files which are in the vectors file.
    FILE * fDataMap;
    errno_t errDataMap;

    if (!file_exists(fn_label_map)) {
        printf("# As you can clearly see, there are none.\n");

        errLabelMap = fopen_s(&fLabelMap, fn_label_map, "wb");
        errDataMap = fopen_s(&fDataMap, fn_data_map, "wb");

        if (errDataMap || errLabelMap) { printf("# Could not open data or label map files!\n");
return -1; }
    }
    else {
        errLabelMap = fopen_s(&fLabelMap, fn_label_map, "r+b");
        errDataMap = fopen_s(&fDataMap, fn_data_map, "r+b");
        if (errDataMap || errLabelMap) { printf("# Could not open data or label map files!\n");
return -1; }

        fseek(fDataMap, 0, SEEK_END);

        /// Every 256bytes are one pair in the dictionary. Starting at byte 0.

        char label_pair[MaximumLabelLength + 2];
        while (fread_s(label_pair, (MaximumLabelLength + 2) * sizeof(char), sizeof(char),
MaximumLabelLength + 1, fLabelMap)) {
            uint8_t label_key = label_pair[0];
            printf("\t> %d | %s\n", label_key, label_pair + 1);
```

```
            existing_labels[existing_labels_count] = label_key;
            existing_labels_count++;
        }

        if (existing_labels_count == 0) {
            printf("# As you can clearly see, there are none.\n");
        }
    }

    fclose(fLabelMap);
    fclose(fDataMap);

    return 0;
}

// Prints the file paths of each file in the dataset sorted by their labels.
int ViewDataFiles() {
    printf("# The following files are in the data set: \n\n");

    uint8_t existing_labels[256];
    uint8_t existing_labels_count = 0;

    // A file with a map (dictionary) of the label numbers (0-255) and their corresponding
"human" definition.
    FILE * fLabelMap;
    errno_t errLabelMap;
    // A file with a map of the actual wav files which are in the vectors file.
    FILE * fDataMap;
    errno_t errDataMap;

    if (!file_exists(fn_data_map)) {
        printf("# As you can clearly see, there are none.\n");

        errLabelMap = fopen_s(&fLabelMap, fn_label_map, "wb");
        errDataMap = fopen_s(&fDataMap, fn_data_map, "wb");

        if (errDataMap || errLabelMap) { printf("# Could not open data or label map files!\n");
return -1; }
    }
    else {
        errLabelMap = fopen_s(&fLabelMap, fn_label_map, "r+b");
        errDataMap = fopen_s(&fDataMap, fn_data_map, "r+b");
        if (errDataMap || errLabelMap) { printf("# Could not open data or label map files!\n");
return -1; }

        int size = getFileSize(fDataMap); fseek(fDataMap, 0, SEEK_SET);
        char * file_data = (char*)malloc(size + 1);
        int read_count = fread_s(file_data, size, sizeof(char), size, fDataMap);
```

```
      if (read_count == 0) {
         printf("# Error reading file.\n"); return -1;
      }


      std::map<uint8_t, char*> label_data_map;

      /// Every 256bytes are one pair in the dictionary. Starting at byte 0.

      char label_pair[MaximumLabelLength + 2];
      while (fread_s(label_pair, (MaximumLabelLength + 2) * sizeof(char), sizeof(char),
MaximumLabelLength + 1, fLabelMap)) {
         uint8_t label_key = label_pair[0];
         //existing_labels[existing_labels_count] = label_key;
         //label_data_map.insert(std::pair<uint8_t,char*>(label_key, label_pair + 1));

         printf("\t> %u | %s\n", label_key, label_pair + 1);

         char * live_data = file_data;

         int pos = 0;
         while (pos < size) {
            uint8_t data_key = live_data[pos];
            if (data_key == label_key) {
               printf("\t|--> %s\n", live_data + pos + 1);
            }
            pos += 2 + strlen(live_data + pos + 1);
         }

         existing_labels_count++;
      }

      if (existing_labels_count == 0) {
         printf("# As you can clearly see, there are none.\n");
      }

      free(file_data);
   }

   fclose(fLabelMap);
   fclose(fDataMap);

   return 0;
}

int ListView(char * command) {
   if (CountCommandArgs(command) > 0) {
      char * arg; GetCommandArg(command, 0, &arg);
```

```
        if (cmpcommand(arg, "-f")) {
            ViewDataFiles();
        }
        else if (cmpcommand(arg, "-c")) {
            ViewClassifications();
        }
        else {
            printf("# Unknown argument!\n");
        }
        free(arg);
        return 0;
    }
    else {
        printf("# Unknown argument!\n");
        return 0;
    }
}

int AddDataToSet() {

    std::string WAV_File_Path;
    std::cout << "# Path for WAV file: " << std::endl;
    std::cout << get_current_dir() << "\\";
    WAV_File_Path = get_current_dir() + "\\";
    char WAV_File_Path_buffer[256];
    fgets(WAV_File_Path_buffer, sizeof(WAV_File_Path_buffer), stdin);
    fix_fgets(WAV_File_Path_buffer);
    for (int i = 0; i < strlen(WAV_File_Path_buffer); i++) {
        WAV_File_Path += WAV_File_Path_buffer[i];
    }
    std::cout << std::endl << std::endl;

    AudioFile<double> * audioFile = new AudioFile<double>();

    if (audioFile->load(WAV_File_Path)) {
        std::cout << "# Loaded File!" << std::endl;
        audioFile->printSummary();
        std::cout << "# CH: " << audioFile->getNumChannels() << " | " << "SAMP: " <<
audioFile->getNumSamplesPerChannel() * audioFile->getNumChannels() << std::endl;

        audioFile->ConcactChannels();

        FFT * FT = new FFT();

        uint8_t channelInd = 0;
        for (int sampInd = 0; sampInd < audioFile->getNumSamplesPerChannel();
sampInd++) {
            FT->AppendToWave(audioFile->samples[channelInd][sampInd]);
```

```
        }
        int actInd = 0;
        std::cout << "\t" << ++actInd << ") " << "Loaded \"" << WAV_File_Path << "\" into
Fourier Transform!" << std::endl;

        uint32_t sample_count = audioFile->getNumSamplesPerChannel() /
audioFile->getSampleRate();

        // A file with a map (dictionary) of the label numbers (0-255) and their corresponding
"human" definition.
        FILE * fLabelMap;
        // A file with a map of the actual wav files which are in the vectors file.
        FILE * fDataMap;

        errno_t errLabelMap = fopen_s(&fLabelMap, fn_label_map, "r+b");
        errno_t errDataMap = fopen_s(&fDataMap, fn_data_map, "r+b");

        if (errDataMap || errLabelMap) { printf("# Could not open Data or Label map
files!\n"); return -1; }

        // Create new vector and label files, or use existing ones and change the sample count
        FILE * fVect;
        FILE * fLabel;
        if (!file_exists(fn_vectors)) {
            errno_t errVects = fopen_s(&fVect, fn_vectors, "wb");
            errno_t errLabels = fopen_s(&fLabel, fn_labels, "wb");
            uint32_t intro[3] = { convert_to_big_endian(2612),
convert_to_big_endian(sample_count), convert_to_big_endian(4000) };
            uint32_t introLabels[2] = { convert_to_big_endian(2211),
convert_to_big_endian(sample_count) };
            fwrite((char*)intro, sizeof(char), 4 * sizeof(uint32_t), fVect);
            fwrite((char*)introLabels, sizeof(char), 2 * sizeof(uint32_t), fLabel);
        }
        else {
            errno_t errVects = fopen_s(&fVect, fn_vectors, "r+b");
            errno_t errLabels = fopen_s(&fLabel, fn_labels, "r+b");
            uint32_t old_sample_count = 0;
            fseek(fLabel, sizeof(uint32_t), SEEK_SET);
            fread_s(&old_sample_count, sizeof(old_sample_count), sizeof(uint32_t), 1,
fLabel);
            old_sample_count = convert_to_little_endian(old_sample_count);
            old_sample_count += sample_count;
            old_sample_count = convert_to_big_endian(old_sample_count);
            fseek(fLabel, sizeof(uint32_t), SEEK_SET);
            fseek(fVect, sizeof(uint32_t), SEEK_SET);
            fwrite(&old_sample_count, sizeof(uint32_t), 1, fLabel);
            fwrite(&old_sample_count, sizeof(uint32_t), 1, fVect);
            fseek(fLabel, 0, SEEK_END);
```

```
        fseek(fVect, 0, SEEK_END);
    }


    printf("Please enter the label (as uint8) for this sound.\n> ");
    char label_buffer[16];
    fgets(label_buffer, sizeof(label_buffer), stdin);
    fix_fgets(label_buffer);

    long lab_long = strtol(label_buffer, NULL, 10);

    uint8_t lab = 0;
    memcpy(&lab, &lab_long, sizeof(uint8_t));
    char label[1] = { lab };


    /*char * label_buffer_temp = (char*)malloc(strlen(label_buffer));
    memcpy(label_buffer_temp, label_buffer, strlen(label_buffer));
    printf("\n\nGot label! %u\n\n", label_buffer_temp[0]);

    sprintf_s(label, sizeof(label), "%u", label_buffer_temp[0]);

    free(label_buffer_temp);*/

    uint8_t existing_labels[256];
    int existing_labels_count = getExistingLabelsCount(existing_labels, fLabelMap);

    fseek(fDataMap, 0, SEEK_END);
    fseek(fLabelMap, 0, SEEK_END);

    fwrite(label, sizeof(uint8_t), 1, fDataMap);
    /*char * WAV_File_Path_c_str = (char*)calloc(WAV_File_Path.length(),
sizeof(char));
    strncpy_s(WAV_File_Path_c_str, WAV_File_Path.length(), WAV_File_Path.c_str(),
WAV_File_Path.length());
    fwrite(WAV_File_Path_c_str, sizeof(char), WAV_File_Path.length(), fDataMap);*/
    fwrite(WAV_File_Path.c_str(), sizeof(char), WAV_File_Path.length(), fDataMap);
    fwrite(OutputPath + 127, sizeof(char), 1, fDataMap);

    bool exists = false;
    if (existing_labels_count > 0) {

        int i = 0;
        while (i < 256 && !exists) {
            exists = exists || existing_labels[i] == lab;
            i++;
        }
    }
```

```
        if (!exists) {
            printf("# This is a new label, please describe it. (in 255 or less characters)\n> ");
            char label_description_buffer[MaximumLabelLength + 1];
            fgets(label_description_buffer, MaximumLabelLength, stdin);
            fix_fgets(label_description_buffer);

            fwrite(label, 1, 1, fLabelMap);
            fwrite(label_description_buffer, sizeof(char), MaximumLabelLength, fLabelMap);

            printf("# Added the pair: {%u, %s} to the label_map.\n", lab,
label_description_buffer);
        }

        printf("# Now loading the file with the label %d.\n", lab);

        /// Create an array of the frequencies
        for (int x = 0; x < sample_count; x++) {

            std::vector<Complex> * liveFreq =
FT->FourierTransfer_Part(audioFile->getSampleRate(), x);

            char * freqArr = (char *)calloc(FT->MaxFrequency(), sizeof(char));

            for (int freqInd = 0; freqInd < FT->MaxFrequency(); freqInd++) {
                freqArr[freqInd] = liveFreq->at(freqInd).real();
            }

            fwrite(freqArr, sizeof(char), FT->MaxFrequency(), fVect);
            fwrite(label, sizeof(char), 1, fLabel);
            free(freqArr);
        }
        fclose(fVect);
        fclose(fLabel);
        fclose(fLabelMap);
        fclose(fDataMap);

        delete FT;

        printf("# Appended sound!\n");
    }

    delete audioFile;

    return 0;
}

// Initialize a new dataset
int Initialize() {
```

```
    FILE * fLabelMap;
    errno_t errLabelMap;
    FILE * fDataMap;
    errno_t errDataMap;
    errLabelMap = fopen_s(&fLabelMap, fn_label_map, "wb");
    errDataMap = fopen_s(&fDataMap, fn_data_map, "wb");
    fclose(fLabelMap);
    fclose(fDataMap);

    return 0;
}

// predict the label of a file using the Classifier
/*
    The predict function can take arguments such as:
    1) a file path.
        The first input can be the relative path to a vectors file which you want to use instead
of processing a wav file into a vectors file.
    2) the '-f' argument.
        This is passed into the classifier as 'fast' which tells the program to only check some of
the data points (arbitrarily) and not all of them.
        This obviously results in faster execution time (10x faster - since we take only a tenth
of the vectors), but there is a penalty to the accuracy of the results. Though the final
prediction is still usually correct.
*/
int Predict(char * command) {
    int argc = CountCommandArgs(command);
    printf("Arg count: %d\n", argc);
    if (argc == 0) {
        char predictOutputFile[256];
        char predictOutputFile_tmp[128];
        printf("Name your prediction file: \n> ");
        fgets(predictOutputFile_tmp, sizeof(predictOutputFile_tmp), stdin);
        fix_fgets(predictOutputFile_tmp);
        sprintf_s(predictOutputFile, sizeof(predictOutputFile), "%s/%s.vectors", OutputPath,
predictOutputFile_tmp);

        std::string WAV_File_Path;
        std::cout << "# Path for WAV file: " << std::endl;
        std::cout << get_current_dir() << "\\";
        WAV_File_Path = get_current_dir() + "\\";
        char WAV_File_Path_buffer[256];
        fgets(WAV_File_Path_buffer, sizeof(WAV_File_Path_buffer), stdin);
        fix_fgets(WAV_File_Path_buffer);
        for (int i = 0; i < strlen(WAV_File_Path_buffer); i++) {
            WAV_File_Path += WAV_File_Path_buffer[i];
        }
        std::cout << std::endl << std::endl;
```

```
    AudioFile<double> * audioFile = new AudioFile<double>();

    // Parse audio file into FFT and export it as vectors into a vectors file
    if (audioFile->load(WAV_File_Path)) {
        std::cout << "# Loaded File!" << std::endl;
        audioFile->printSummary();
        std::cout << "# CH: " << audioFile->getNumChannels() << " | " << "SAMP: " <<
audioFile->getNumSamplesPerChannel() * audioFile->getNumChannels() << std::endl;

        FFT * FT = new FFT();

        uint8_t channelInd = 0;
        for (int sampInd = 0; sampInd < audioFile->getNumSamplesPerChannel();
sampInd++) {
            FT->AppendToWave(audioFile->samples[channelInd][sampInd]);
        }
        int actInd = 0;
        std::cout << "\t" << ++actInd << ") " << "Loaded \"" << WAV_File_Path << "\"
into Fourier Transform!" << std::endl;

        uint32_t sample_count = audioFile->getNumSamplesPerChannel() /
audioFile->getSampleRate();

        FILE * fVect;
        errno_t errVects;
        if (!file_exists(predictOutputFile)) {
            errVects = fopen_s(&fVect, predictOutputFile, "wb");
            uint32_t intro[3] = { convert_to_big_endian(2612),
convert_to_big_endian(sample_count), convert_to_big_endian(4000) };
            fwrite((char*)intro, sizeof(char), 3 * sizeof(uint32_t), fVect);
        }
        else {
            errVects = fopen_s(&fVect, predictOutputFile, "r+b");
            uint32_t old_sample_count = 0;
            fread_s(&old_sample_count, sizeof(old_sample_count), sizeof(uint32_t), 1,
fVect);
            old_sample_count = convert_to_little_endian(old_sample_count);
            old_sample_count += sample_count;
            old_sample_count = convert_to_big_endian(old_sample_count);
            fseek(fVect, sizeof(uint32_t), SEEK_SET);
            fwrite(&old_sample_count, sizeof(uint32_t), 1, fVect);
            fseek(fVect, 0, SEEK_END);
        }
        if (errVects) { printf("Could not find nor create prediction file!\n"); return -1; }

        /// Create an array of the frequencies
        for (int x = 0; x < sample_count; x++) {
```

```
            std::vector<Complex> * liveFreq =
FT->FourierTransfer_Part(audioFile->getSampleRate(), x);

            char * freqArr = (char *)calloc(FT->MaxFrequency(), sizeof(char));

            for (int freqInd = 0; freqInd < FT->MaxFrequency(); freqInd++) {
               freqArr[freqInd] = liveFreq->at(freqInd).real();
            }

            fwrite(freqArr, sizeof(char), FT->MaxFrequency(), fVect);
            free(freqArr);
         }
         fclose(fVect);

         delete FT;
      }
      delete audioFile;

      printf("# Created prediction file!!\n");

      data_handler->read_predict_feature_vector(predictOutputFile);
   }
   else if (argc == 1) {
      char * predictOutputFile;
      if (GetCommandArg(command, 0, &predictOutputFile) != 0) { printf("# Could not
find argument!\n"); return -1; }
      data_handler->read_predict_feature_vector(predictOutputFile);
   }
   else {
      printf("# Unrecognized argument count!\n");
   }

   data_handler->normalize_prediction_data();

   for (int i = 0; i < data_handler->get_prediction_data_size(); i++) {
      data_handler->get_prediction_data()->at(i)->c_only();
   }

   network->set_prediction_data(data_handler->get_prediction_data());

   int prediction_output = network->real_predict();


   FILE * fLabelMap;
   errno_t errLabelMap = fopen_s(&fLabelMap, fn_label_map, "r+b");
   if (errLabelMap) { printf("# Could not open label map file!\n"); return -1; }
```

```
    char label_pair[MaximumLabelLength + 2];
    while (fread_s(label_pair, (MaximumLabelLength + 2) * sizeof(char), sizeof(char),
MaximumLabelLength + 1, fLabelMap)) {
        uint8_t label_key = label_pair[0];
        if (label_key == prediction_output) {
            printf("The prediction for this file is: \n\t%d | %s\n", prediction_output, label_pair +
1);
        }
    }

    fclose(fLabelMap);


    network->set_training_data(data_handler->get_training_data());
    network->set_validation_data(data_handler->get_validation_data());
    network->set_test_data(data_handler->get_test_data());

    return 0;
}

int InitNetwork(char * command) {
    if (data_handler == NULL) {
        printf("You must have a data_handler before initializing a network!\n");
        return 2;
    }

    printf("Now initializing network of model: ");
    printf("%d -> ", data_handler->get_training_data()->at(0)->get_feature_array_size());
    std::vector<int> hidden_layers;
    for (int i = 0; i < CountCommandArgs(command); i++) {
        char * arg;
        GetCommandArg(command, i, &arg);
        hidden_layers.push_back(strtol(arg, NULL, 10));
        printf("%d -> ", hidden_layers.at(i));
    }
    printf("%d\n", data_handler->get_class_count());


    network = new Network(
        hidden_layers,
        data_handler->get_training_data()->at(0)->get_feature_array_size(),
        data_handler->get_class_count(),
        0.25);

    network->set_training_data(data_handler->get_training_data());
    network->set_validation_data(data_handler->get_validation_data());
    network->set_test_data(data_handler->get_test_data());
```

```
    network->c_only();

    return 0;
}

int LoadNetwork(char * command) {
    if (network != NULL) {
        printf("You currently have a network with an accuracy of %.4f%% loaded. Please
dereference it before proceeding!\n", network->test_performance * 100.0);
        return 2;
    }

    FILE * f;
    errno_t err = fopen_s(&f, command, "rb");
    if (err == 0) {
        // Successfully opened file for reading

        /// IMPORT
        unsigned long long file_size = 0;
        fread(&file_size, sizeof(file_size), 1, f);
        fseek(f, 0, SEEK_SET);
        char * buffer_read = (char*)calloc(file_size, sizeof(char));

        uint32_t read = fread(buffer_read, sizeof(char), file_size, f);

        fclose(f);

        network = new Network(0.25);
        network->import_network(buffer_read);
        free(buffer_read);

        network->set_training_data(data_handler->get_training_data());
        network->set_validation_data(data_handler->get_validation_data());
        network->set_test_data(data_handler->get_test_data());


    }
    else {
        printf("Failed to open or read file '%s' which was requested as a network import file!",
command);
    }

    return 0;
}

int ExportNetwork(char * command) {
    printf("Network test performance: %.4f%%\n", 100.0*network->test_c());
```

```
    char * buffer;
    int total_size = network->export_network(&buffer);

    FILE * f;
    errno_t err = fopen_s(&f, command, "wb");
    fwrite(buffer, sizeof(char), total_size, f);
    fclose(f);

    return 0;
}

int TrainNetwork(char * command) {
    if (network == NULL) {
        printf("You must first load or initialize a network before training it!\n");
        return 2;
    }

    long epochs = 0;
    epochs = strtol(command, NULL, 10);

    printf("Now training %d times...\n", epochs);

    for (int i = 0; i < epochs; i++) {
        printf("Training error @ iteration %d: %.4f\n", i, network->train_c());
    }

    return 0;
}

int ThreadTrainNetwork(char * command) {
    if (network == NULL) {
        printf("You must first load or initialize a network before training it!\n");
        return 2;
    }

    long epochs = 0;
    epochs = strtol(command, NULL, 10);

    printf("Now training %d times...\n", epochs);

    auto lamba = [&](int target) {
        for (int i = 0; i < epochs; i++) {
            printf("Thread #%d => Training error @ iteration %d: %.4f\n", target, i,
network->train_c(target));
        }
    };

    std::vector<std::thread> threads;
```

```
    for (int target = 0; target < data_handler->get_class_count(); target++)
    {
        threads.emplace_back(std::thread(lamba, target));
    }

    for (auto &th : threads)
    {
        th.join();
    }

    return 0;
}

int AFKTrain() {
    std::vector<std::thread> threads;

    auto lambdaTrain = [&]() {
        double last_validation_performance = 0;

        printf("Started training thread!\nType: 'stop' to exit!\n");

        while (1) {

            for (int epoch = 0; epoch < 5; epoch++) {
                network->train_c();
            }

            double validation_performance = network->validate_c();

            printf("Validation performance: %.4f%%\n", 100.0 * validation_performance);

            if (validation_performance > last_validation_performance) {
                char export_network_file_name[64] = { 0 };
                sprintf_s(export_network_file_name, 64, "NetworkOutput_%.2f.net", 100.0 *
validation_performance);
                ExportNetwork(export_network_file_name);
            }

        }
    };

    auto lambdaAwaitExit = [&](std::thread * training_thread) {
        while (1) {
            char buffer[64];
            fgets(buffer, 64, stdin);
            fix_fgets(buffer);
```

```
            if (cmpcommand(buffer, "stop")) {
                (*training_thread).join();
                break;
            }
        }
    };

    std::thread training_thread = std::thread(lambdaTrain);
    std::thread exit_thread = std::thread(lambdaAwaitExit, &training_thread);

    exit_thread.join();

    return 0;
}

int ValidateNetwork() {
    if (network == NULL) {
        printf("You must first load or initialize a network before validating it!\n");
        return 2;
    }

    network->set_training_data(data_handler->get_training_data());
    network->set_validation_data(data_handler->get_validation_data());
    network->set_test_data(data_handler->get_test_data());

    printf("Validation performance: %.4f%%\n", 100.0 * network->validate_c());

    return 0;
}

int TestNetwork() {
    if (network == NULL) {
        printf("You must first load or initialize a network before validating it!\n");
        return 2;
    }

    network->set_training_data(data_handler->get_training_data());
    network->set_validation_data(data_handler->get_validation_data());
    network->set_test_data(data_handler->get_test_data());

    printf("Test performance: %.4f%%\n", 100.0 * network->test_c());

    return 0;
}

int LoadDataHandler() {
    if (data_handler != NULL) { delete data_handler; }
    data_handler = new Data_Handler();
```

```
    data_handler->read_feature_vector(fn_vectors);
    data_handler->read_feature_labels(fn_labels);
    data_handler->count_classes();
    data_handler->normalize_data();
    data_handler->split_data();
    data_handler->c_only();

    return 0;
}
int ResetDataHandler() {
    if (data_handler != NULL) { delete data_handler; }
    data_handler = new Data_Handler();

    return 0;
}
int LoadPredictFile(char * command) {
    if (data_handler == NULL) {
        printf("There is no data_handler, please initialize one before loading a predict
file!\n");
        return 2;
    }

    data_handler->read_predict_feature_vector(command);
    data_handler->normalize_prediction_data();

    for (int i = 0; i < data_handler->get_prediction_data_size(); i++) {
        data_handler->get_prediction_data()->at(i)->c_only();
    }

    return 0;
}

int ExitProgram() {
    free(fn_data_map);
    free(fn_label_map);
    free(fn_labels);
    free(fn_vectors);
    exit(0);
    return 0;
}
int ShowNoHelpMenu(bool ext) {
    printf("Please select a help menu: \n");
    const char * menus[6] = { "This menu", "General", "Data Set", "Predictions", "Neural
Network", "All" };
    for (int i = 0; i < 6; i++) {
        printf("\t%d) for %s help type '%d'\n", i, menus[i], i);
    }
    printf("\tTo exit type '-1'\n");
```

```
    int helpMenuToShow = -1;
    char inp[16];
    fgets(inp, 16, stdin);
    fix_fgets(inp);

    helpMenuToShow = strtol(inp, NULL, 10);

    if (helpMenuToShow == -1) { return 0; }
    else {
        Help((HelpMenus)helpMenuToShow);
    }
}
int ShowGeneralHelpMenu(bool ext) {
    printf("> General Help Menu\n");
    printf("$ help {0} | To see help menu.\n");
    printf("\t {0} => The help menu to show as an integer (type 'help 0' for more...).\n");
    printf("$ exit | Exits the program.\n");
    printf("$ dsinit | DataSet Initialize. Initializes a new dataset. You must do this when
starting a new dataset!\n");
    printf("$ cds | Change to a different dataset in the same directory.\n");
    printf("$ cdir | Change output directory.\n");
    return 0;
}
int ShowDataSetHelpMenu(bool ext) {
    printf("> DataSet Help Menu\n");

    printf("$ list {0} | List: \n");
    printf("\t -c | Classification - classes. Show a list of the classes which are labeled in the
dataset.\n");
    printf("\t -f | Files. Show a list of the files which are in the dataset.\n");

    printf("$ add | Add a file to the dataset\n");

    printf("$ dhload | Data Handler Load. Loads the dataset into the live data handler (HDD
=> RAM). Do this before training or testing (or validating) a network.\n");
    printf("$ dhreset | Unloades a loaded dataset from the data handler, or creates a blank
data handler.\n");

    return 0;
}
int ShowPredictionHelpMenu(bool ext) {
    printf("> Prediction Help Menu\n");

    printf("$ pred {0} | Predict.\n");
    printf("\t{0} => Prediction file path | If a prediction file (A vectors file) already exists,
you may use this to instead of creating a new prediction file.\n");
```

```
    return 0;
}
int ShowNetworkHelpMenu(bool ext) {
    printf("> Neural Network Help Menu\n");

    printf("$ ninit {...} | Initialize a new neural network.\n");
    printf("\tEach argument should be the size of a hidden layer for the new neural
network.");
    printf("\t'ninit 120 120 32' will create a neural network with 5 layers: an input layer
(automatically created acording to the datahandler), 3 hidden layers of size: 120, 120, 32,
and an output layer (also automatic).\n");

    printf("$ ntrain {0} | Train a network a certain amount of times. Default = 1\n");
    printf("$ nthtrain {0} | Network threaded train. Train a network on multiple threads a
certain amount of epochs. Warning: Multithread training may harm the network, use only
with an untrained network.\n");
    printf("$ nafktrain | Train a network and export it until stopped.\n");

    printf("$ nvalid | Validate the network.\n");
    printf("$ ntest | Test the network.\n");

    printf("$ nexport {0} | Export the network. {0} => File path and name.\n");
    printf("$ nimport {0} | Import a network. {0} => File path and name.\n");

    return 0;
}

int Help(HelpMenus helpMenu) {
    bool ext = false;

    switch (helpMenu)
    {
    case NoHelpMenu:
        ShowNoHelpMenu(ext);
        break;
    case GeneralHelpMenu:
        ShowGeneralHelpMenu(ext);
        break;
    case DataSetHelpMenu:
        ShowDataSetHelpMenu(ext);
        break;
    case PreidctionHelpMenu:
        ShowPredictionHelpMenu(ext);
        break;
    case NetworkHelpMenu:
        ShowNetworkHelpMenu(ext);
        break;
    case AllHelpMenu:
```

```
        ShowGeneralHelpMenu(ext);
        ShowDataSetHelpMenu(ext);
        ShowPredictionHelpMenu(ext);
        ShowNetworkHelpMenu(ext);
        break;
    default:
        ShowNoHelpMenu(ext);
        break;
    }

    return 0;
}
int Help(char * command) {
    HelpMenus helpMenu = NoHelpMenu;

    if (CountCommandArgs(command) > 0) {
        helpMenu = (HelpMenus)strtol(command, NULL, 10);
    }

    return Help(helpMenu);
}

// Gets user input and compares it to valid commands
int ProccessCommands() {
    printf("> ");
    char input[1024];
    fgets(input, sizeof(input), stdin);
    fix_fgets(input);

    if (cmpcommand(input, "help")) { return Help(input + strlen("help") + 1); }
    if (cmpcommand(input, "exit") || cmpcommand(input, "close")) { return ExitProgram();
}

    if (cmpcommand(input, "dsinit")) { return Initialize(); }
    if (cmpcommand(input, "cds")) { return ChangeDataSetName(); }
    if (cmpcommand(input, "cdir")) { return ChangeOutputPath(); }

    if (cmpcommand(input, "list")) { return ListView(input + strlen("list") + 1); }

    if (cmpcommand(input, "add")) { return AddDataToSet(); }

    if (cmpcommand(input, "dhload")) { return LoadDataHandler(); }
    if (cmpcommand(input, "dhreset")) { return ResetDataHandler(); }

    if (cmpcommand(input, "pred")) { return Predict(input + strlen("pred") + 1); }

    if (cmpcommand(input, "ninit")) { return InitNetwork(input + strlen("ninit") + 1); }
    if (cmpcommand(input, "ntrain")) { return TrainNetwork(input + strlen("ntrain") + 1); }
```

```
    if (cmpcommand(input, "nafktrain")) { return AFKTrain(); }
    if (cmpcommand(input, "nthtrain")) { return ThreadTrainNetwork(input +
strlen("nthtrain") + 1); }
    if (cmpcommand(input, "nvalid")) { return ValidateNetwork(); }
    if (cmpcommand(input, "ntest")) { return TestNetwork(); }

    if (cmpcommand(input, "nimport")) { return LoadNetwork(input + strlen("nimport") +
1); }
    if (cmpcommand(input, "nexport")) { return ExportNetwork(input + strlen("nexport") +
1); }

    return 1;
}
```

AudioClassifier.cpp (main)

```
#include "CommandLineFunctions.h"

#include <chrono>
#include <utility>
typedef std::chrono::high_resolution_clock::time_point TimeVar;

#define duration(a) std::chrono::duration_cast<std::chrono::nanoseconds>(a).count()
#define timeNow() std::chrono::high_resolution_clock::now()

int StartUp() {
    printf("\n");
    printf("  _____                    _  _____ _                 _  __ _                \n");
    printf("  / ____|                  | |/ ____| |               (_)/ _(_)               \n");
    printf(" | (___   ___  _   _ _ __   __| | |    | | __ _ ___ ___ _| |_ _  ___ _ __ \n");
    printf("  \\___ \\ / _ \\| | | | '_ \\ / _` | |    | |/ _` / __/ __| |  _| |/ _ \\ '__|\n");
    printf("  ____) | (_) | |_| | | | | (_| | |____| | (_| \\__ \\__ \\ | | | |  __/ |   \n");
    printf(" |_____/ \\___/ \\__,_|_| |_|\\__,_|  \\_____|_|\\__,_|___/___/_|_| |_|\\___|_|   \n");
    printf("                                                                            \n");

    printf("Copyright © 2021 Michael Kuperfish Steinberg\n\n\n");

    ChangeOutputPath();
    ChangeDataSetName();

    while (true) {
        if (ProccessCommands() == 1) {
            printf("# Command Failed!\n");
        }
    }
```

```
    return 0;
}

int main(int argc, char ** argv)
{
    return StartUp();
}
```

Works Cited

"10." *Shutterstock*, ak6.picdn.net/shutterstock/videos/1027713866/thumb/10.jpg.

"17.7 - Calling Inherited Functions and Overriding Behavior." *Learn C++*, www.learncpp.com/cpp-tutorial/calling-inherited-functions-and-overriding-behavior/.

"17.8 - Hiding Inherited Functionality." *Learn C++*, www.learncpp.com/cpp-tutorial/hiding-inherited-functionality/.

About Thorben JanssenThorben is an independent trainer and author of the Amazon bestselling book Hibernate Tips - More than 70 solutions to common Hibernate problems.He writes about Java EE related topics on his blog Thoughts on Java. "OOP Concept for Beginners: What Is Inheritance?" *Stackify*, 2 June 2020, stackify.com/oop-concept-inheritance/.

Adamstark. "Adamstark/AudioFile." *GitHub*, github.com/adamstark/AudioFile.

Aquarius_GirlAquarius_Girl 18.1k5656 gold badges186186 silver badges346346 bronze badges, et al. "How to Print a Char Array in C through Printf?" *Stack Overflow*, 1 Mar. 1967, stackoverflow.com/questions/50312194/how-to-print-a-char-array-in-c-through-printf.

"Artificial Neural Network." *Wikipedia*, Wikimedia Foundation, 22 Mar. 2021, en.wikipedia.org/wiki/Artificial_neural_network.

Atul, Kang &, et al. "Snake Game with Deep Learning." *TheAILearner*, 12 Dec. 2018, theailearner.com/2018/04/19/snake-game-with-deep-learning/.

"Audio File Format Specifications." *Wave File Specifications*, www-mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/WAVE.html.

Bettilyon, Tyler Elliot. “How to Classify MNIST Digits with Different Neural Network Architectures.” *Medium*, Teb's Lab, 9 May 2019, medium.com/tebs-lab/how-to-classify-mnist-digits-with-different-neural-network-architectures-39c75a0f03e3.

Braunschweig, Dave. “Encapsulation.” *Programming Fundamentals*, 15 Dec. 2018, press.rebus.community/programmingfundamentals/chapter/encapsulation/#:~:text=Encapsulation%20is%20one%20of%20the,parties'%20direct%20access%20to%20them.

“Breakout (Video Game).” *Infogalactic*, 1 Apr. 1976, infogalactic.com/info/Breakout_(video_game).

“Bytes to Integer - C++ Forum.” *Cplusplus.com*, www.cplusplus.com/forum/beginner/3076/.

The C++ Resources Network. “Data Structures.” *Cplusplus.com*, www.cplusplus.com/doc/tutorial/structures/.

“C++.” *Wikipedia*, Wikimedia Foundation, 21 Mar. 2021, en.wikipedia.org/wiki/C%2B%2B.

Cerna, Michael, and Audrey F. Hervey. “The Fundamentals of FFT-Based Signal Analysis and Measurement.” National Instruments, July 2000.

“Common Misconceptions about Birds - Arctic and Antarctic Birds.” *Beyond Penguins and Polar Bears*, beyondpenguins.ehe.osu.edu/issue/arctic-and-anarctic-birds/common-misconceptions-about-birds.

Corob-Msft. “x64 Calling Convention.” *Microsoft Docs*, docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-160.

Darkshots6692. “The Ultimate Eminem Piano Medley.” *YouTube*, YouTube, 8 Sept. 2015, www.youtube.com/watch?v=BJp_16j8s-A.

DiCola, Tony. “FFT: Fun with Fourier Transforms.” *Adafruit Learning System*, learn.adafruit.com/fft-fun-with-fourier-transforms/background.

“Discrete Fourier Transform.” *Wikipedia*, Wikimedia Foundation, 17 Mar. 2021, en.wikipedia.org/wiki/Discrete_Fourier_transform.

“Fast Fourier Transform.” *Fast Fourier Transform - Rosetta Code*, rosettacode.org/wiki/Fast_Fourier_transform.

“Fast Fourier Transform.” *Wikipedia*, Wikimedia Foundation, 8 Mar. 2021, en.wikipedia.org/wiki/Fast_Fourier_transform.

Fourier Analysis and Synthesis, hyperphysics.phy-astr.gsu.edu/hbase/Audio/fourier.html.

“Frequency Ranges of Musical Instruments: Musical Instruments, Musicals, Music Theory.” *Pinterest*, www.pinterest.com/pin/469641067374231384/.

“Function - Dictionary Definition.” *Vocabulary.com*, www.vocabulary.com/dictionary/function.

gerardonfiya. “C++ Machine Learning Tutorial Part 3: K-Means Clustering Unsupervised Learning.” *YouTube*, YouTube, 6 Feb. 2019, www.youtube.com/watch?v=knXGdIUExBY.

Godbolt, Matt. *Compiler Explorer*, godbolt.org/.

“Gradient Descent Derivation.” *Gradient Descent Derivation · Chris McCormick*, 4 Mar. 2014, mccormickml.com/2014/03/04/gradient-descent-derivation/.

Henrik Hillestad LøvoldHenrik Hillestad Løvold 1, et al. “Reading File as Hex in C++.” *Stack Overflow*, 1 Oct. 1962, stackoverflow.com/questions/20336810/reading-file-as-hex-in-c.

“Hidden Markov Model.” *Wikipedia*, Wikimedia Foundation, 24 Mar. 2021, en.wikipedia.org/wiki/Hidden_Markov_model.

"How Many Dog Breeds Are There?: Hill's Pet." *Hill's Pet Nutrition*, www.hillspet.com/dog-care/behavior-appearance/how-many-dog-breeds-are-there.

"Information Hiding." *Wikipedia*, Wikimedia Foundation, 28 Mar. 2021, en.wikipedia.org/wiki/Information_hiding.

"Inheritance (Object-Oriented Programming)." *Wikipedia*, Wikimedia Foundation, 11 Feb. 2021, en.wikipedia.org/wiki/Inheritance_(object-oriented_programming).

"Inheritance, Polymorphism and the Object Memory Model." Cs.bgu.ac.il, 2010.

Karthikeyan, Vijayanarasimhan. "How Can We Differentiate between People's Voices? Why Are People's Voices Different?" *Quora*, 9 Nov. 2015, www.quora.com/How-can-we-differentiate-between-peoples-voices-Why-are-peoples-voices-different.

Katharina von Kriegstein, Anne-Lise Giraud. "Implicit Multisensory Associations Influence Voice Recognition." *PLOS Biology*, Public Library of Science, journals.plos.org/plosbiology/article?id=10.1371%2Fjournal.pbio.0040326.

L, Noah. "Sharing Objects Between Threads in C++, the Safe and Easy Way." *CodeProject*, CodeProject, 14 June 2016, www.codeproject.com/Articles/1106491/Sharing-Objects-Between-Threads-in-Cplus plus-the-S.

Liashchynskyi, Petro. "Creating of Neural Network Using JavaScript in 7 Minutes!" *DEV Community*, DEV Community, 12 Jan. 2019, dev.to/liashchynskyi/creating-of-neural-network-using-javascript-in-7minutes-o21.

"Marin Mersenne." *Wikipedia*, Wikimedia Foundation, 18 Mar. 2021, en.wikipedia.org/wiki/Marin_Mersenne.

"MNIST Database." *Wikipedia*, Wikimedia Foundation, 10 Mar. 2021, en.wikipedia.org/wiki/MNIST_database.

Monson, Brian B., et al. "The Perceptual Significance of High-Frequency Energy in the Human Voice." *Frontiers*, Frontiers, 26 May 2014, www.frontiersin.org/articles/10.3389/fpsyg.2014.00587/full.

"My Shazam Adventure." *luungoc2005*, 16 Jan. 2017, luungoc2005.github.io/posts/tutorials/dotnet/naudio/my-shazam-adventure/.

"Neural Circuit." *Wikipedia*, Wikimedia Foundation, 8 Mar. 2021, en.wikipedia.org/wiki/Neural_circuit.

"Object-Oriented Programming." *Wikipedia*, Wikimedia Foundation, 22 Mar. 2021, en.wikipedia.org/wiki/Object-oriented_programming.

officialtiesto. "Tiësto & Sevenn - BOOM (Official Video)." *YouTube*, YouTube, 26 Apr. 2017, www.youtube.com/watch?v=tSJSVmfaMCs.

"Online x86 / x64 Assembler and Disassembler." ▼, defuse.ca/online-x86-assembler.htm.

oscaralsing. "What Is The Difference Between KNN and K-Means?" *YouTube*, YouTube, 25 Jan. 2018, www.youtube.com/watch?v=OClrEI_5Ri4.

Packt Subscription, subscription.packtpub.com/book/game_development/9781849699099/1/ch01lvl1sec09/analog-and-digital-audio.

"Procedural Programming." *Wikipedia*, Wikimedia Foundation, 15 Feb. 2021, en.wikipedia.org/wiki/Procedural_programming.

Puzzle Magazine. "Minesweeper Puzzle Magazine." *Puzzle Magazines*, www.puzzle-magazine.com/minesweeper-magazine.php.

Reed, Smith Alexander. “THE MUSICAL SEMIOTICS OF TIMBRE IN THE HUMAN VOICE and STATIC TAKES LOVE’S BODY .” *University of Pittsburgh* , 2005.

“Research Guides: MLA Style: In-Text Citations.” *In-Text Citations - MLA Style - Research Guides at Towson University*, towson.libguides.com/mlastyle/in-text.

Roberts, Bucky, director. *C++ Programming Tutorials*. *YouTube*, TheNewBoston, 11 Apr. 2011, www.youtube.com/playlist?list=PLAE85DE8440AA6B83.

Shalev-Shwartz, Shai, and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2017.

Smith, Keith. “Blockade [Model 807-0001].” *Return to the Mainpage*, Gaming-History (Arcade-History), 2011, www.arcade-history.com/?n=blockade&page=detail&id=287.

“Spectrum Analyzer.” *Spectrum Analyzer | Academo.org - Free, Interactive, Education.*, academo.org/demos/spectrum-analyzer/.

“Std::Cout, Std::Wcout.” *Cppreference.com*, en.cppreference.com/w/cpp/io/cout.

“Std::Vector::Size.” *Cplusplus.com*, www.cplusplus.com/reference/vector/vector/size/.

Umamaheswaran, Venkatesh. “Comprehending K-Means and KNN Algorithms.” *Medium*, Becoming Human: Artificial Intelligence Magazine, 14 Nov. 2018, becominghuman.ai/comprehending-k-means-and-knn-algorithms-c791be90883d.

Wang, Avery Li-Chun. “An Industrial-Strength Audio Search Algorithm .” Shazam Entertainment, Ltd. .

Wang, Avery Li-Chun. “An Industrial-Strength Audio Search Algorithm.” Shazam Entertainment, Ltd., 2003.

What Is a Static Polymorphism in C#?, www.tutorialspoint.com/What-is-a-static-polymorphism-in-Chash.

"What Is Backpropagation Really Doing? | Deep Learning, Chapter 3." *YouTube*, YouTube, 3 Nov. 2017, www.youtube.com/watch?v=Ilg3gGewQ5U.

"What Is the Toupper() Function in C?" *Educative*, www.educative.io/edpresso/what-is-the-toupper-function-in-c.

"X86 Calling Conventions." *Wikipedia*, Wikimedia Foundation, 24 Mar. 2021, en.wikipedia.org/wiki/X86_calling_conventions.