portfolio_1

Henry Bourne

2023-01-30

Intro to C++

In this document we will go over the basics of C++, the idea of this document is for it to act as a quick reference sheet to refresh on C++ syntax should I forget. We will use bash to print out and run C++ files all of which (along with this R markdown file can be found on my github: "github.com/h-aze/compass_yr1/SC2"). Some explanation I will give before the code chunks and some I will give in the comments of the C++ files.

The basics

We will start by covering the basics which is all contained in the "intro.cpp" file, lets print it out and run it:

```
cat intro_to_cpp/intro.cpp
```

```
## #include <iostream>
## #include <string>
## // We will explain what the above mean shortly
##
## // This chunk of code describes how to define a function aswell as some useful syntax,
## // read this once you have gotten to where a_function is called in main!
## // Again everything in c++ is typed including functions
## // if we didn't want to return anything we could set the type to "void"
## double a_function(int arg)
## {
##     // We can condition using the following syntax and print out something useful about arg
##     if (arg <0){
##         std::cout << arg << " is negative." << std::endl;
##     }
##     else if (arg < 20){
##         std::cout << arg << " is between 0 and 20." << std::endl;
##     }
##     else{
##         std::cout << arg << " is bigger than 20" << std::endl;
##     }
##
##     // We can carry out a for loop as follows which prints out the numbers from 10 to 1
##     // We could also write i=i-1 as i-- or i-=1
##     for (int i=10; i>0; i=i-1){
##         std::cout << i << std::endl;
##     }
##
##     double out = arg * 0.01;
##     return out;
## }
```

```
##
## // We now define a function main and put all our code inside,
## // everything inside main will then be run when we run this code
## int main()
## {
##     // In C++ everything is typed and so when we define anything we must make sure we type it,
##     // eg. we can define a variable that is the integer 1 as follows:
##     int a = 1 ;
##
##     // Notice that we put a ";" after as this is how c++ defines line breaks,
##     // if we wanted to define a string we could do the following:
##     std::string our_string = "Hello!";
##     std::cout << our_string << std::endl;
##
##     // Here we define a variable our_string that is of type string (std::string)
##     // and then print the output.
##     // We use "#include <string>" to include the string header file
##     // which allows us to use the string type.
##     // We also type "#include <iostream>" which gives us the functionality
##     // of getting inputs and giving outputs,
##     // for example if we want to print to the console we need to include this header file.
##     // We can then use std::cout and std::endl as above to print to the console.
##     // Other things we can do with this header file is throw exceptions and return errors,
##     // we will write how to do this below but comment them out
##     // so we don't get any errors when running this code:
##     // std::cerr << "Some error" << std::endl;
##     // throw std::runtime_error("Unknown exception");
##
##     // We now call the function "a_function" which we define above.
##     // Go read that portion of code now!
##     a_function(10) ;
##
##     return 0;
## }
```

```
g++ intro_to_cpp/intro.cpp -o intro_to_cpp/intro
./intro_to_cpp/intro
```

```
## Hello!
## 10 is between 0 and 20.
## 10
## 9
## 8
## 7
## 6
## 5
## 4
## 3
## 2
## 1
```

Types

Let's now discuss types in more detail:

```
cat intro_to_cpp/types.cpp
```

```
## // We first include header files we will need in this script
## #include <iostream>
## #include <string>
##
## int main(){
##     // Let's first define a variable of type double
##     double var = 10.5;
##     std::cout << "Var as a double: " << var << std::endl;
##
##     // We can change the type of the variable as follows:
##     int var1 = var;
##     std::cout << "Var now converted to int: " << var1 << std::endl;
##
##     // If you check the output of running this file
##     // you can see we have lost information
##     // So must be careful when converting types
##
##     // We also have auto:
##     // if it is obvious what type a var or fun should have,
##     // you can use "auto" eg,
##     auto var2 = var1 ;
##     // ^ Here it is "obvious" that var2 should be of type int
##     std::cout << "Var assigned using auto: " << var2 << std::endl;
## }
```

```
g++ intro_to_cpp/types.cpp -o intro_to_cpp/types
./intro_to_cpp/types
```

```
## Var as a double: 10.5
## Var now converted to int: 10
## Var assigned using auto: 10
```

Scope

Scope is controlled by "{}", anything defined in the brackets is only available in the brackets. It is also defined by files, anything defined in a file is only available in that file, unless we import of course (we will discuss this later). Note that if we have multiple definitions of a variable in different levels of nests we use the definition of the inner most nest we currently are in.

Vectors and Dictionaries

```
cat intro_to_cpp/vector_dict.cpp
```

```
## // Don't read this for now ...
## // --------------------------------------
## #include <iostream>
## #include <vector> // We need this to use vectors
## #include <map> // We need this to use dictionaries (maps in C++)
##
## // This function creates a vector of vectors (a matrix)
## std::vector<std::vector<int>> get_nested_vector(){
##     std::vector< std::vector<int> > M;
```

```
##
##     // We create a vector of vectors
##     // ie. a matrix
##     for (int i=1; i<=3; ++i)
##     {
##         // We first create a row
##         std::vector<int> row;
##
##         for (int j=1; j<=3; ++j)
##         {
##             row.push_back( i * j );
##         }
##
##         // Now we save the row to the matrix
##         M.push_back(row);
##     }
##     return M;
## }
##
## // ----------------------------------------
## // START READING HERE!
## int main(){
##     // First we will talk vectors,
##     // a vector is a container that can store multiple values of only one type
##
##     // We can create a vector that can hold integers as so:
##     std::vector<int> v;
##
##     // We can add values onto the end as so:
##     v.push_back( 4 );
##     v.push_back( 2.5 );
##     // Note for the second command 2.5 will be converted to an integer
##     // Let's now print out our vector,
##     // note we use a range-based for loop here (needs C++11 or later)
##     std::cout << "Our vector: { " ;
##     for (auto i: v)
##         std::cout << i << ' ';
##     std::cout << "}" << std::endl;
##
##     // We can get the size of the vector
##     int length = v.size();
##     std::cout << "The vector has length: " << length << std::endl;
##
##     // Is also possible to nest vectors inside vectors
##     auto M = get_nested_vector();
##     // go to the get_nested_vector to see how we create and fill a matrix!
##     std::cout << "A nested vector: " << std::endl;
##     // this prints the matrix out
##     for (int i=0; i<3; ++i)
##     {
##         for (int j=0; j<3; ++j)
##         {
##             std::cout << M[i][j] << " ";
##         }
```

```
##
##         std::cout << std::endl;
##     }
##
##     // ----------
##     // Now lets talk dictionaries,
##     // dictionaries in C++ are called maps,
##     // they are containers that store key value pairs,
##     // note that keys must be of the same type and values must be of the same type
##
##     // We create a map that stores strings and where the keys are strings
##     std::map<std::string, std::string> dict;
##
##     // we can add some items to the map
##     dict["key_1"] = "value_1";
##     dict["key_2"] = "value_2";
##
##     // Now we can loop through all of the key-value pairs
##     // in the map and print them out
##     for ( auto item : dict )
##     {
##         //item.first is the key
##         std::cout << "Getting the key using .first :"
##         << item.first << " , ";
##
##         //item.second is the value
##         std::cout << "Getting the key using .second :"
##         << item.second << std::endl;
##     }
##
##     // Finally we can lookup values by key
##     std::cout << "What's the value associated to key_1?: " << dict["key_1"]
##               << std::endl;
##
##
##     return 0;
## }
```

```
g++ intro_to_cpp/vector_dict.cpp -o intro_to_cpp/vector_dict
./intro_to_cpp/vector_dict
```

```
## Our vector: { 4 2 }
## The vector has length: 2
## A nested vector:
## 1 2 3
## 2 4 6
## 3 6 9
## Getting the key using .first :key_1 , Getting the key using .second :value_1
## Getting the key using .first :key_2 , Getting the key using .second :value_2
## What's the value associated to key_1?: value_1
```

Multi-File Programmes

In C++ we use something called a header file (uses the ".h" extension). Header files are where you store all your function declarations and in the .cpp files you should store all your function definitions and main code.

What this achieves is a separation between interface (the header file) and implementation (the .cpp file). To use the function declarations in your cpp file you must use "#include". If we then want to use a function that was declared in a header file but defined elsewhere all we must do is "#include" the header file, this is how we use functions from standard libraries. There are two ways of "#include"-ing:

```
# include <standard_library_name>

# include 'user_defined_library'
```

Another thing to mention that we haven't yet is that you can overload functions, ie. you can create multiple instances of a function that take different types and at run time the definition of the function that has the correct types and number of arguments is used. Before moving on we will mention header guards, we must include these to prevent the situation where we copy are code in twice (if we have two includes of the same thing in our code for example), at the top of the file one should write:

```
# ifndef _FILENAME_H

# define _FILENAME_H
```

and at the bottom:

```
# endif
```

Finally we should have a file named "main.cpp" where we keep our main function.

Objects, Classes, Concepts, Default Arguments and Operators

Finally we will talk briefly about objects, classes, concepts, default arguments and operators. We can declare a class in C (in the header file!) using the following syntax:

```
#'class ClassName
#'{
#'public:
#'    ClassName(type arg); //This is the constructor (or initialization) method
#'
#'    type class_method(type arg);
#'
#'private:
#'    type an_attribute;
#'};
```

We can then define a method we have declared in a .cpp file using the following syntax:

```
#'ClassName::name_of_class_method(type arg)
#'{
#'  do something
#'}
```

If we want to change the value of an attribute we can do it as follows:

```
#'this->attribute_name = value;
```

We can initialize a class as follows:

```
#'ClassName obj(init_param);
```

And then use a method as follows:

```
#'obj.method_name(params);
```

Now we will talk about operators. For classes in C++ we can define operators, operators are functions that are added to classes to specify what code should be used when we operate on them with other classes. Examples of key operators are:

```
# ;operator+ : addition ;operator- : subtraction ;operator*
# : multiplication ;operator/ : division ;bool operator== :
# comparison equals to, ;bool operator!= : not equal to
# ;bool operator< : less than ;bool operator<= : less than
# or equal to ;bool operator> : greater than ;bool
# operator>= : greater than or equal to
```

We can declare an operator for a class like so:

```
#'ClassName operator+(ClassName arg_name);
```

And then define it like so:

```
#'ClassName ClassName::operator+(ClassName arg_name)
#'{
#'    do and return something
#'}
```

This concludes the intro to C++!

Let's Get Some C++ Practice!

To get coding some more C++ we will now move onto a project I have done in C++. The project was to create, train and test a neural network all written in C++. I wrote the neural network completely from scratch using only the standard C++ libraries in order to get some good C++ practice and to get more familiar with some of the details of how a neural network works.

Creating a Neural Network

The neural network class is defined in the "Neural_Net.h" file, we will now print out the header file so we can get an idea of what the class looks like:

```
cat neural_network/Neural_Net.h
```

```
## #ifndef NEURALNETWORK_H
## #define NEURALNETWORK_H
##
## #include <vector>
## #include <string>
## #include <memory>
##
## // Function for computing the MSE between the output and target vectors
## double computeMSE(const std::vector<double>& output, const std::vector<double>& target);
##
## // Function for creating a random double number between min and max
## double randomDouble(double min, double max);
##
## // Function to initialize a matrix of random weights according to arguments
## std::vector<std::vector<double>> initializeWeights(size_t numRows, size_t numCols, double min, doubl
##
## // Function to initialize a vector of random biases according to arguments
## std::vector<double> initializeBiases(size_t numBiases, double min, double max);
##
```

```
## // Virtual class which all activation function classes should implement
## class ActivationFunction {
## public:
##     virtual double activate(double x) const = 0;
##     virtual double derivative(double x) const = 0;
##     virtual std::string get_name() const = 0;
## };
##
## // Class for the ReLU activation function, it inherits from the virtual ActivationFunction class
## class ReLU : public ActivationFunction {
## public:
##     double activate(double x) const override;
##     double derivative(double x) const override;
##     std::string get_name() const override;
## };
##
## // Layer class
## class Layer {
## private:
##     std::vector<std::vector<double>> weights;
##     std::vector<double> biases;
##     const std::shared_ptr<ActivationFunction> actFun;
##     std::vector<double> preActivationValues;
##     // Store the gradients of the weights and biases for all the data points in the current batch
##     std::vector<std::vector<std::vector<double>>> weightGradients;
##     std::vector<std::vector<double>> biasGradients;
##
## public:
##     Layer(const std::vector<std::vector<double>>& initialWeights,
##             const std::vector<double>& initialBiases,
##             const std::shared_ptr<ActivationFunction>& actFun
##         );
##     Layer(int n_in,
##             int n_out,
##             const std::shared_ptr<ActivationFunction>& actFun
##         );
##     ~Layer();
##
##     std::vector<double> layer_pass(const std::vector<double>& inputs);
##
##     std::vector<double> get_activation_derivative() const;
##
##     std::vector<std::vector<double>> get_weights() const;
##
##     std::vector<double> get_biases() const;
##
##     std::shared_ptr<ActivationFunction> get_actFun() const;
##
##     std::vector<double> get_preActivationValues() const;
##
##     std::vector<std::vector<std::vector<double>>> get_weightGradients() const;
##
##     std::vector<std::vector<double>> get_biasGradients() const;
##
```

```
##     void set_weights(const std::vector<std::vector<double>>& newWeights);
##
##     void set_biases(const std::vector<double>& newBiases);
##
##     void add_weightGradients(const std::vector<std::vector<double>>& newWeightGradients);
##
##     void add_biasGradients(const std::vector<double>& newBiasGradients);
##
##     void clear_weightGradients();
##
##     void clear_biasGradients();
##
##     void print() const;
## };
##
## // Neural Network class
## class NeuralNetwork {
## private:
##     std::vector<Layer> layers;
##
## public:
##     NeuralNetwork(const std::vector<Layer>& layers);
##     ~NeuralNetwork();
##
##     std::vector<double> forwardPass(const std::vector<double>& input);
##
##     std::vector<double> computeLossGradient(const std::vector<double>& output, const std::vector<doul
##
##     std::vector<std::vector<double>> computeWeightGradients(const std::vector<double>& delta, const l
##
##     std::vector<double> computeBiasGradients(const std::vector<double>& delta) const;
##
##     void updateWeightsAndBiases(Layer& layer, const std::vector<std::vector<double>>& weightGradients
##
##     std::vector<double> computeDelta(const std::vector<double>& delta, const Layer& layer) const;
##
##     double backpropagation(const std::vector<std::vector<double>>& input, const std::vector<std::vec
##
##     void print() const;
## };
##
## #endif // NEURALNETWORK_H
```

We can see that the class has a constructor, a destructor, a method to forward propogate through the network and a method to back propogate through the network for a mini-batch of datapoints (the network is trained using mini-batch gradient descent). There is also the layer class which contains methods for forward and back propogation through the individual layer which the neural network class makes use of. For now the only activation function we have defined is the ReLU function and the only kind of layers we have defined are fully connected layers. We also currently can only use the MSE as our loss and so in future it would be good to add cross entropy loss for when carrying out classification tasks for example. However, it should be straightforward to add more activation functions, loss functions and layer types such as convolutional layers.

Let's now look at the "Optimizer.h" file:

```
cat neural_network/Optimizer.h
```

```
## // Optimizer.h
## #ifndef OPTIMIZER_H
## #define OPTIMIZER_H
##
## #include <vector>
## #include <string>
## #include <memory>
## #include <iostream>
## #include <numeric>
## #include <random>
## #include "Neural_Net.h"
##
## // Optimizer class
## template <typename DT, typename LT>
## class Optimizer {
## private:
##     NeuralNetwork nn;
##     const std::vector<DT> trainingData;
##     const std::vector<LT> trainingLabels;
##     const std::vector<DT> testData;
##     const std::vector<LT> testLabels;
##     const std::vector<DT> validationData;
##     const std::vector<LT> validationLabels;
##     const std::string opt_method;
##     const double learningRate;
##     const int batchSize;
##     const int numEpochs;
##     const int numBatches;
##     std::vector<int> trainingInd;
##
## public:
##     Optimizer(NeuralNetwork& nn,
##             const std::vector<DT>& trainingData,
##             const std::vector<LT>& trainingLabels,
##             const std::vector<DT>& testData,
##             const std::vector<LT>& testLabels,
##             const std::vector<DT>& validationData,
##             const std::vector<LT>& validationLabels,
##             const std::string opt_method = "SGD",
##             const double learningRate = 0.01,
##             const int batchSize = 32,
##             const int numEpochs = 10);
##
##     Optimizer(NeuralNetwork& nn,
##             const std::vector<DT>& trainingData,
##             const std::vector<LT>& trainingLabels,
##             const std::vector<DT>& testData,
##             const std::vector<LT>& testLabels,
##             const std::string opt_method = "SGD",
##             const double learningRate = 0.01,
##             const int batchSize = 32,
##             const int numEpochs = 10);
```

```
##
##     ~Optimizer();
##
##     // Gets a batch of data and labels from the training data
##     std::pair<std::vector<DT>, std::vector<LT>> getTrainBatch(const int& batchNum) const;
##
##     // Gets all the training data
##     std::pair<std::vector<DT>, std::vector<LT>> getTrainData() const;
##
##     // Gets a batch of data and labels from the validation data
##     std::pair<std::vector<DT>, std::vector<LT>> getValidation() const;
##
##     // Gets a batch of data and labels from the test data
##     std::pair<std::vector<DT>, std::vector<LT>> getTest() const;
##
##     // Shuffles the training data and labels
##     void shuffleTrainData();
##
##     // Performs one epochs worth of training, returns the average loss on each batch
##     std::vector<double> singleTrain();
##
##     // Performs a single testing step, ie. forward pass and loss calculation for whole test set and r
##     double singleTest();
##
##     // Iteratively conducts training step for every batch and for number of specified epochs
##     void train();
##
##     NeuralNetwork getNn() const;
##
##     std::vector<int> getTrainingInd() const;
##
## };
##
## #include "Optimizer.tpp"
##
## #endif // OPTIMIZER_H
```

We can see that the optimizer class has a constructor, a destructor, methods to retrieve training, testing and validation data, a method to train the network and a method to test the network among other things. The Optimizer class makes use of templates so that it can work with any kind of data. We can use the train() method to train a network on some data for a given number of epochs, for each epoch it will train the network on all the data and then test the network on the test data printing the loss. Some work remains to be done to log the loss and accuracies for each epoch and plot them, fully implement the use of validation data (at the moment it just uses the test data) and implement early stopping, learning rate schedules, etc.

The way we could create a network is instantiate a vector of layers and initialize the neural network class with it. We could then instantiate an optimizer with some data and our neural network and train the network. My hope was to create a wrapper for the classes with Rcpp so that we could use the neural network in R and try out the network on some data! but I have not quite had the time to do so. I am fairly confident, however, that the code should work (or at least be very close to working) as I have written extensive unit tests for the code using google test, which my code passes. In the appendix you can see all the tests for yourself, but for now lets just run the tests:

```
cd neural_network
make
```

```
./runTests
```

```
## [100%] Built target runTests
## [==========] Running 40 tests from 7 test suites.
## [----------] Global test environment set-up.
## [----------] 2 tests from RandomDoubleTest
## [ RUN      ] RandomDoubleTest.PositiveValues
## [       OK ] RandomDoubleTest.PositiveValues (0 ms)
## [ RUN      ] RandomDoubleTest.NegativeValues
## [       OK ] RandomDoubleTest.NegativeValues (0 ms)
## [----------] 2 tests from RandomDoubleTest (0 ms total)
##
## [----------] 2 tests from InitializeWeightsTest
## [ RUN      ] InitializeWeightsTest.DifferentSizes
## [       OK ] InitializeWeightsTest.DifferentSizes (0 ms)
## [ RUN      ] InitializeWeightsTest.MinBiggerThanMax
## [       OK ] InitializeWeightsTest.MinBiggerThanMax (0 ms)
## [----------] 2 tests from InitializeWeightsTest (0 ms total)
##
## [----------] 3 tests from initializeBiasesTest
## [ RUN      ] initializeBiasesTest.PositiveValues
## [       OK ] initializeBiasesTest.PositiveValues (0 ms)
## [ RUN      ] initializeBiasesTest.ZeroValues
## [       OK ] initializeBiasesTest.ZeroValues (0 ms)
## [ RUN      ] initializeBiasesTest.InvalidValues
## [       OK ] initializeBiasesTest.InvalidValues (0 ms)
## [----------] 3 tests from initializeBiasesTest (0 ms total)
##
## [----------] 3 tests from ReLUClassTest
## [ RUN      ] ReLUClassTest.Activate
## [       OK ] ReLUClassTest.Activate (0 ms)
## [ RUN      ] ReLUClassTest.Derivative
## [       OK ] ReLUClassTest.Derivative (0 ms)
## [ RUN      ] ReLUClassTest.GetName
## [       OK ] ReLUClassTest.GetName (0 ms)
## [----------] 3 tests from ReLUClassTest (0 ms total)
##
## [----------] 12 tests from LayerTest
## [ RUN      ] LayerTest.Constructor
## [       OK ] LayerTest.Constructor (0 ms)
## [ RUN      ] LayerTest.ConstructorWithValidInputs
## [       OK ] LayerTest.ConstructorWithValidInputs (0 ms)
## [ RUN      ] LayerTest.LayerPass
## [       OK ] LayerTest.LayerPass (0 ms)
## [ RUN      ] LayerTest.SetWeightsPositiveTestCase
## [       OK ] LayerTest.SetWeightsPositiveTestCase (0 ms)
## [ RUN      ] LayerTest.SetWeightsNegativeTestCase
## [       OK ] LayerTest.SetWeightsNegativeTestCase (0 ms)
## [ RUN      ] LayerTest.GetBiases
## [       OK ] LayerTest.GetBiases (0 ms)
## [ RUN      ] LayerTest.GetBiasesEmpty
## [       OK ] LayerTest.GetBiasesEmpty (0 ms)
## [ RUN      ] LayerTest.SetBiasesPositive
## [       OK ] LayerTest.SetBiasesPositive (0 ms)
```

```
## [ RUN      ] LayerTest.SetBiasesNegative
## [       OK ] LayerTest.SetBiasesNegative (0 ms)
## [ RUN      ] LayerTest.ActivationDerivativeTest
## [       OK ] LayerTest.ActivationDerivativeTest (0 ms)
## [ RUN      ] LayerTest.InvalidWeightsSize
## [       OK ] LayerTest.InvalidWeightsSize (0 ms)
## [ RUN      ] LayerTest.InvalidInputsSize
## [       OK ] LayerTest.InvalidInputsSize (0 ms)
## [----------] 12 tests from LayerTest (0 ms total)
##
## [----------] 11 tests from NeuralNetworkTest
## [ RUN      ] NeuralNetworkTest.ForwardPass
## [       OK ] NeuralNetworkTest.ForwardPass (0 ms)
## [ RUN      ] NeuralNetworkTest.InvalidLayersSize
## [       OK ] NeuralNetworkTest.InvalidLayersSize (0 ms)
## [ RUN      ] NeuralNetworkTest.ComputeLossPositive
## [       OK ] NeuralNetworkTest.ComputeLossPositive (0 ms)
## [ RUN      ] NeuralNetworkTest.ComputeLossNegative
## [       OK ] NeuralNetworkTest.ComputeLossNegative (0 ms)
## [ RUN      ] NeuralNetworkTest.ComputeWeightGradients_PositiveTest
## [       OK ] NeuralNetworkTest.ComputeWeightGradients_PositiveTest (0 ms)
## [ RUN      ] NeuralNetworkTest.ComputeWeightGradients_NegativeTest
## [       OK ] NeuralNetworkTest.ComputeWeightGradients_NegativeTest (0 ms)
## [ RUN      ] NeuralNetworkTest.ComputeBiasGradients
## [       OK ] NeuralNetworkTest.ComputeBiasGradients (0 ms)
## [ RUN      ] NeuralNetworkTest.UpdateWeightsAndBiases_ValidInput
## [       OK ] NeuralNetworkTest.UpdateWeightsAndBiases_ValidInput (0 ms)
## [ RUN      ] NeuralNetworkTest.updateWeightsAndBiasesNegativeTest
## [       OK ] NeuralNetworkTest.updateWeightsAndBiasesNegativeTest (0 ms)
## [ RUN      ] NeuralNetworkTest.ComputeDeltaTest
## [       OK ] NeuralNetworkTest.ComputeDeltaTest (0 ms)
## [ RUN      ] NeuralNetworkTest.ComputeDeltaNegativeTest
## [       OK ] NeuralNetworkTest.ComputeDeltaNegativeTest (0 ms)
## [----------] 11 tests from NeuralNetworkTest (0 ms total)
##
## [----------] 7 tests from OptimizerTest
## [ RUN      ] OptimizerTest.getTrainBatchTest
## [       OK ] OptimizerTest.getTrainBatchTest (0 ms)
## [ RUN      ] OptimizerTest.getValidationTest
## [       OK ] OptimizerTest.getValidationTest (0 ms)
## [ RUN      ] OptimizerTest.getTestTest
## [       OK ] OptimizerTest.getTestTest (0 ms)
## [ RUN      ] OptimizerTest.shuffleTrainData
## [       OK ] OptimizerTest.shuffleTrainData (1 ms)
## [ RUN      ] OptimizerTest.singleTrainTest
## Batch 0
## [       OK ] OptimizerTest.singleTrainTest (0 ms)
## [ RUN      ] OptimizerTest.singleTestTest
## Test loss: 1
## [       OK ] OptimizerTest.singleTestTest (0 ms)
## [ RUN      ] OptimizerTest.trainTest
## Epoch 0
## Batch 0
## Test loss: 0.9802
```

```
## Epoch 1
## Batch 0
## Test loss: 0.961184
## Average training loss for each batch:
## 0.5
## 0.4901
## Average test loss for each epoch:
## 0.4901 0.480592
## [       OK ] OptimizerTest.trainTest (0 ms)
## [----------] 7 tests from OptimizerTest (1 ms total)
##
## [----------] Global test environment tear-down
## [==========] 40 tests from 7 test suites ran. (1 ms total)
## [  PASSED  ] 40 tests.
```

Further work with this code that I would also like to finish at some point is paralleling the training of the network to speed it up along with other things mentioned earlier. After this project my C++ knowledge certainly feels much sounder although its a shame I couldn't quite round the whole thing off by interfacing with Rcpp and training it on some data! But I perhaps bit off more than I can chew given the time frame! The next section I have printed off some of the implementation code, but again, please check my Github as it may be easier to read there.

Appendix: The implementations of the header files and the unit tests

I will now print out the implementations of the header files if you are interested in taking a look at how all the methods were implemented:

```
cat neural_network/Neural_Net.cpp
```

```
## #include "Print_Utils.h"
## #include "Utils.h"
## #include "Neural_Net.h"
## #include <iostream>
## #include <random>
## #include <string>
## #include <memory>
## #include <numeric>
##
## // Implementations for initialization related functions
## // ---------------------------------------------------------------------
##
## // Function for computing the MSE between the output and target vectors
## double computeMSE(const std::vector<double>& output, const std::vector<double>& target) {
##     if (output.size() != target.size()) {
##         throw std::runtime_error("Vector dimensions do not match.");
##     }
##
##     size_t size = output.size();
##     double sumSquaredError = 0.0;
##
##     for (size_t i = 0; i < size; i++) {
##         double error = output[i] - target[i];
##         sumSquaredError += error * error;
##     }
##
```

```
##     return sumSquaredError / static_cast<double>(size);
## }
##
## // Function to generate a random double value within a specified range
## double randomDouble(double min, double max) {
##     std::random_device rd;
##     std::mt19937 gen(rd());
##     std::uniform_real_distribution<> dis(min, max);
##     return dis(gen);
## }
##
## // Function to initialize a matrix of weights with random values
## std::vector<std::vector<double>> initializeWeights(size_t numRows, size_t numCols, double min, double
##     // Check that the number of rows and columns is greater than 0
##     if (numRows == 0 || numCols == 0) {
##         throw std::invalid_argument("Number of rows and columns must be greater than 0");
##     }
##     // Check that the min value is less than the max value
##     if (min >= max) {
##         throw std::invalid_argument("Min value must be less than max value");
##     }
##     // Initialize the matrix of weights with random values
##     std::vector<std::vector<double>> weights(numRows, std::vector<double>(numCols));
##     for (size_t i = 0; i < numRows; ++i) {
##         for (size_t j = 0; j < numCols; ++j) {
##             weights[i][j] = randomDouble(min, max);
##         }
##     }
##     return weights;
## }
##
## // Function to initialize a vector of random biases according to arguments
## std::vector<double> initializeBiases(size_t numBiases, double min, double max){
##     // Check that the number of biases is greater than 0
##     if (numBiases == 0) {
##         throw std::invalid_argument("Number of biases must be greater than 0");
##     }
##     // Check that the min value is less than the max value
##     if (min >= max) {
##         throw std::invalid_argument("Min value must be less than max value");
##     }
##     // Initialize the vector of biases with random values
##     std::vector<double> biases(numBiases);
##     for (size_t i = 0; i < numBiases; ++i) {
##         biases[i] = randomDouble(min, max);
##     }
##     return biases;
## }
##
## // Implementations for ActivationFunction virtual class
## // ------------------------------------------------------------------
##
## // Class for the ReLU activation function, it inherits from the virtual ActivationFunction class
## double ReLU::activate(double x) const {
```

```
##     double out ;
##     if(x>0){
##         out = x;
##     }
##     else{
##         out = 0;
##     }
##     return out;
## }
##
## double ReLU::derivative(double x) const {
##     double out ;
##     if(x>0){
##         out = 1;
##     }
##     else{
##         out = 0;
##     }
##     return out;
## }
##
## std::string ReLU::get_name() const{
##     return "ReLU";
## }
##
## // Implementation for Layer class
## // -------------------------------------------------------------------------
##
## // Constructor for the Layer class when we have a matrix of weights we want to use
## // Layer::Layer(const std::vector<std::vector<double>>& initialWeights, const std::shared_ptr<Activa
## //     weights(initialWeights), actFun(actFun) {}
##
## Layer::Layer(const std::vector<std::vector<double>>& initialWeights, const std::vector<double>& initi
##     : weights(initialWeights), biases(initialBiases), actFun(actFun), preActivationValues(initialWeig
## {
##     if (isMatrix(weights) == false) {
##         throw std::invalid_argument("Weights must be a matrix");
##     }
##     if (isVector(biases) == false) {
##         throw std::invalid_argument("Biases must be a vector");
##     }
##     // Check that the number of biases is the same as the number of rows in the weights matrix
##     if (biases.size() != weights.size()) {
##         throw std::invalid_argument("Number of biases must be the same as the number of rows in the w
##     }
## }
##
## // Constructor for the Layer class when we want to randomly initialize the weights
## Layer::Layer(int n_in, int n_out, const std::shared_ptr<ActivationFunction>& actFun):
##     weights(initializeWeights(n_out, n_in, -1, 1)), biases(initializeBiases(n_out, -1, 1)), actFun(ac
##
## // Deconstructor for the Layer class
## Layer::~Layer(){}
##
```

```
## // Computes the product of the input with the matrix of weights and adds the bias for the layer and p
## std::vector<double> Layer::layer_pass(const std::vector<double>& inputs) {
##     // Check the inputs is a vector and not empty or a matrix
##     if(isVector(inputs) == false){
##         throw std::invalid_argument("Inputs must be a vector");
##     }
##     std::vector<double> out(weights.size());
##     std::vector<double> out_pre(weights.size()) ;
##     for (size_t i = 0; i < weights.size(); ++i){
##         double sum = 0;
##         for (size_t j = 0; j < inputs.size(); ++j){
##             sum += weights[i][j] * inputs[j];
##         }
##         sum += biases[i];
##         preActivationValues[i] = sum;
##         out[i] = actFun->activate(sum);
##     }
##     return out;
## };
##
## // Computes the derivative of the activation function for the layer
## std::vector<double> Layer::get_activation_derivative() const{
##     std::vector<double> out ;
##     for (size_t i = 0; i < preActivationValues.size(); ++i){
##         out.push_back(actFun->derivative(preActivationValues[i]));
##     }
##     return out;
## };
##
## // Returns the weights for the layer
## std::vector<std::vector<double>> Layer::get_weights() const{
##     return weights;
## };
##
## // Returns the biases for the layer
## std::vector<double> Layer::get_biases() const{
##     return biases;
## };
##
## // Returns the activation function for the layer
## std::shared_ptr<ActivationFunction> Layer::get_actFun() const{
##     return actFun;
## };
##
## // Returns the values of neurons before being passed through the activation function
## std::vector<double> Layer::get_preActivationValues() const{
##     return preActivationValues;
## };
##
## // Returns all stored weight gradients for the layer
## std::vector<std::vector<std::vector<double>>> Layer::get_weightGradients() const{
##     return weightGradients;
## };
##
```

```
## // Returns all stored bias gradients for the layer
## std::vector<std::vector<double>> Layer::get_biasGradients() const{
##     return biasGradients;
## };
##
## // Sets the weights for the layer
## void Layer::set_weights(const std::vector<std::vector<double>>& newWeights){
##     // Check that the new weights matrix has the same dimensions as the current weights matrix
##     if (newWeights.size() != weights.size() || newWeights[0].size() != weights[0].size()) {
##         throw std::invalid_argument("New weights matrix must have the same dimensions as the current
##     }
##
##     weights = newWeights;
## };
##
## // Sets the biases for the layer
## void Layer::set_biases(const std::vector<double>& newBiases){
##     // Check that the new biases vector has the same dimensions as the current biases vector
##     if (newBiases.size() != biases.size()) {
##         throw std::invalid_argument("New biases vector must have the same dimensions as the current l
##     }
##
##     biases = newBiases;
## };
##
## // Add a new set of weight gradients
## void Layer::add_weightGradients(const std::vector<std::vector<double>>& newWeightGradients){
##     // Check that the new weight gradients matrix has the same dimensions as the current weight gradi
##     if (newWeightGradients.size() != weightGradients.size() || newWeightGradients[0].size() != weight
##         throw std::invalid_argument("New weight gradients matrix must have the same dimensions as the
##     }
##
##     weightGradients.push_back(newWeightGradients);
## };
##
## // Add a new set of bias gradients
## void Layer::add_biasGradients(const std::vector<double>& newBiasGradients){
##     // Check that the new bias gradients vector has the same dimensions as the current bias gradients
##     if (newBiasGradients.size() != biasGradients.size()) {
##         throw std::invalid_argument("New bias gradients vector must have the same dimensions as the c
##     }
##
##     biasGradients.push_back(newBiasGradients);
## };
##
## // Clears all stored weight gradients for the layer
## void Layer::clear_weightGradients(){
##     weightGradients.clear();
## };
##
## // Clears all stored bias gradients for the layer
## void Layer::clear_biasGradients(){
##     biasGradients.clear();
## };
```

```
##
## // Prints the weights and activation function for the layer
## void Layer::print() const {
##     std::cout << "Weights:" << std::endl;
##     printMatrix(weights);
##
##     std::cout << "Activation Function: ";
##     if (actFun) {
##         std::cout << actFun->get_name() << std::endl;
##     } else {
##         std::cout << "None" << std::endl;
##     }
## };
##
## // Implementation for Neural Network class
## // -----------------------------------------------------------------------
## // Constructor for the Neural Network class
## NeuralNetwork::NeuralNetwork(const std::vector<Layer>& layers): layers(layers) {
##     // Check the dimension of layer sizes match
##     for (size_t i = 0; i < layers.size() - 1; ++i) {
##         if (layers[i].get_weights().size() != layers[i + 1].get_weights()[0].size()) {
##             throw std::invalid_argument("Layer sizes must match");
##         }
##     }
## }
##
## // Deconstructor for the Neural Network class
## NeuralNetwork::~NeuralNetwork(){}
##
## // Computes a forward pass through the network
## std::vector<double> NeuralNetwork::forwardPass(const std::vector<double>& input){
##     std::vector<double> out = input;
##     for(size_t i=0; i < layers.size(); ++i){
##         out = layers[i].layer_pass(out);
##     }
##     return out;
## };
##
## // Computes the gradient of the loss of the output of the network
## std::vector<double> NeuralNetwork::computeLossGradient(const std::vector<double>& output, const std:
##     // Check that the output and target vectors have the same size
##     if (output.size() != target.size()) {
##         throw std::invalid_argument("Output and target sizes must match");
##     }
##
##     std::vector<double> lossGradient(output.size());
##     for (size_t i = 0; i < output.size(); ++i) {
##         // Compute the derivative of the MSE loss with respect to the output
##         lossGradient[i] = 2.0 * (output[i] - target[i]);
##     }
##
##     return lossGradient;
## };
##
```

```
## // Computes the gradients of the weights for a layer
## std::vector<std::vector<double>> NeuralNetwork::computeWeightGradients(const std::vector<double>& del
##     // Check that the delta vector and the weights matrix have the same number of rows
##     if (delta.size() != layer.get_weights().size()) {
##         throw std::invalid_argument("Delta and weights sizes must match");
##     }
##
##     // Compute the gradients for the weights
##     std::vector<std::vector<double>> weightGradients(layer.get_weights().size(), std::vector<double>
##     for (size_t i = 0; i < weightGradients.size(); ++i) {
##         for (size_t j = 0; j < weightGradients[0].size(); ++j) {
##             weightGradients[i][j] = delta[i] * layer.get_weights()[i][j];
##         }
##     }
##
##     return weightGradients;
## };
##
## // Computes the gradients of the biases for a layer
## std::vector<double> NeuralNetwork::computeBiasGradients(const std::vector<double>& delta) const {
##     // Compute the gradients for the biases
##     std::vector<double> biasGradients(delta.size());
##     for (size_t i = 0; i < biasGradients.size(); ++i) {
##         biasGradients[i] = delta[i];
##     }
##
##     return biasGradients;
## };
##
## // Updates the weights and biases for a layer
## void NeuralNetwork::updateWeightsAndBiases(Layer& layer, const std::vector<std::vector<double>>& weig
##     // Check that the weight gradients and the weights matrix have the same dimensions
##     if (weightGradients.size() != layer.get_weights().size() || weightGradients[0].size() != layer.ge
##         throw std::invalid_argument("Weight gradients and weights sizes must match");
##     }
##
##     // Check that the bias gradients and the weights matrix have matching dimensions
##     if (biasGradients.size() != layer.get_weights().size()) {
##         throw std::invalid_argument("Bias gradients and weights sizes must match");
##     }
##     // Set new weights
##     auto new_weights = layer.get_weights();
##     for (size_t i = 0; i < layer.get_weights().size(); ++i) {
##         for (size_t j = 0; j < layer.get_weights()[0].size(); ++j) {
##             new_weights[i][j] -= learningRate * weightGradients[i][j];
##         }
##     }
##     layer.set_weights(new_weights);
##     // Set new biases
##     auto new_biases = layer.get_biases();
##     for (size_t i = 0; i < layer.get_biases().size(); ++i) {
##         new_biases[i] -= learningRate * biasGradients[i];
##     }
##     layer.set_biases(new_biases);
```

```
## };
##
## // Computes the delta for a layer
## std::vector<double> NeuralNetwork::computeDelta(const std::vector<double>& delta, const Layer& layer)
##     // Check that the delta vector and the weights matrix have the same number of rows
##     if (delta.size() != layer.get_weights().size()) {
##         throw std::invalid_argument("Delta and weights sizes must match");
##     }
##
##     // Compute the delta for the previous layer
##     std::vector<double> previousDelta(layer.get_weights()[0].size());
##     std::vector<double> act_deriv = layer.get_activation_derivative();
##     for (size_t i = 0; i < previousDelta.size(); ++i) {
##         double sum = 0.0;
##         for (size_t j = 0; j < delta.size(); ++j) {
##             sum += delta[j] * layer.get_weights()[j][i];
##         }
##         previousDelta[i] = sum * act_deriv[i];
##     }
##
##     return previousDelta;
## };
##
## // For a mini-batch of data-points backpropagates the loss and updates the weights and biases accordi
## double NeuralNetwork::backpropagation(const std::vector<std::vector<double>>& inputs, const std::vec
##     // Initialize containers for the gradients of the weights and biases
##     std::vector<std::vector<std::vector<std::vector<double>>>> minibatchWeightGradients(inputs.size()
##     std::vector<std::vector<std::vector<double>>> minibatchBiasGradients(inputs.size(), std::vector<s
##     // Initialize container for the loss
##     double avg_loss = 0.0;
##     // Loop to calculate the loss and gradients for each data point in the batch
##     for(size_t i=0; i < inputs.size(); ++i){
##         // Get the input and target for the current data point
##         std::vector<double> input = inputs[i];
##         std::vector<double> target = targets[i];
##
##         // Perform forward pass to get the output
##         std::vector<double> output = forwardPass(input);
##
##         // Compute the Loss Gradient
##         std::vector<double> delta = computeLossGradient(output, target);
##
##         // Keep track of the loss
##         avg_loss += computeMSE(output, target);
##
##         // Loop to calculate the gradients for each layer
##         for (int j = layers.size() - 1; j >= 0; --j) {
##             Layer& layer = layers[j];
##
##             // Compute gradients for weights and biases
##             minibatchWeightGradients[i][j] = computeWeightGradients(delta, layer);
##             minibatchBiasGradients[i][j] = computeBiasGradients(delta);
##
##             // Compute gradients for previous layer
```

```
##             delta = computeDelta(delta, layer);
##         }
##     }
##
##     for (int i = layers.size() - 1; i >= 0; --i) {
##         std::vector<std::vector<double>> avgWeightGradients(minibatchWeightGradients[0][i].size(),st
##         std::vector<double> avgBiasGradients(minibatchBiasGradients[0][i].size(), 0);
##         // For current layer i, get the gradients for the weights and biases for all the data points
##         for (size_t j = 0; j < minibatchWeightGradients.size(); ++j) {
##             avgWeightGradients = matrixAddition(avgWeightGradients, minibatchWeightGradients[j][i]);
##             avgBiasGradients = vectorAddition(avgBiasGradients, minibatchBiasGradients[j][i]);
##         }
##         matrixScalarMultiplication(avgWeightGradients, 1.0 / minibatchWeightGradients.size());
##         vectorScalarMultiplication(avgBiasGradients, 1.0 / minibatchBiasGradients.size());
##         // Get the layer
##         Layer& layer = layers[i];
##         // Update weights and biases
##         updateWeightsAndBiases(layer, avgWeightGradients, avgBiasGradients, learningRate);
##     }
##     return avg_loss / inputs.size();
## };
##
## // Prints information on all the layers in the network
## void NeuralNetwork::print() const {
##     std::cout << "Neural Network with following layers{" << std::endl;
##     for(const auto& l : layers){
##         std::cout << "Layer:" << std::endl;
##         l.print();
##         std::cout << "\n" << std::endl;
##     }
##     std::cout << "}" << std::endl;
## };
```

```
cat neural_network/Optimizer.tpp
```

```
## // Optimizer constructor and destructor
## // ------------------------------------------------------------------------
##
## // Constructor with validation data
## template <typename DT, typename LT>
## Optimizer<DT, LT>::Optimizer(NeuralNetwork& nn,
##         const std::vector<DT>& trainingData,
##         const std::vector<LT>& trainingLabels,
##         const std::vector<DT>& testData,
##         const std::vector<LT>& testLabels,
##         const std::vector<DT>& validationData,
##         const std::vector<LT>& validationLabels,
##         const std::string opt_method,
##         const double learningRate,
##         const int batchSize,
##         const int numEpochs
##     ) : nn(nn),
##         trainingData(trainingData),
##         trainingLabels(trainingLabels),
##         testData(testData),
```

```
##         testLabels(testLabels),
##         validationData(validationData),
##         validationLabels(validationLabels),
##         opt_method(opt_method),
##         learningRate(learningRate),
##         batchSize(batchSize),
##         numEpochs(numEpochs),
##         numBatches(trainingData.size() / batchSize),
##         trainingInd(trainingData.size())
##     {
##         if (trainingData.size() != trainingLabels.size()) {
##             throw std::invalid_argument("Training data and labels must be the same size");
##         }
##         if (testData.size() != testLabels.size()) {
##             throw std::invalid_argument("Test data and labels must be the same size");
##         }
##         if (validationData.size() != validationLabels.size()) {
##             throw std::invalid_argument("Validation data and labels must be the same size");
##         }
##         std::iota(trainingInd.begin(), trainingInd.end(), 0);
## };
##
## // Constructor without validation data
## template <typename DT, typename LT>
## Optimizer<DT, LT>::Optimizer(NeuralNetwork& nn,
##         const std::vector<DT>& trainingData,
##         const std::vector<LT>& trainingLabels,
##         const std::vector<DT>& testData,
##         const std::vector<LT>& testLabels,
##         const std::string opt_method,
##         const double learningRate,
##         const int batchSize,
##         const int numEpochs
##     ) : nn(nn),
##         trainingData(trainingData),
##         trainingLabels(trainingLabels),
##         testData(testData),
##         testLabels(testLabels),
##         opt_method(opt_method),
##         learningRate(learningRate),
##         batchSize(batchSize),
##         numEpochs(numEpochs),
##         numBatches(trainingData.size() / batchSize),
##         trainingInd(trainingData.size())
##     {
##         if (trainingData.size() != trainingLabels.size()) {
##             throw std::invalid_argument("Training data and labels must be the same size");
##         }
##         if (testData.size() != testLabels.size()) {
##             throw std::invalid_argument("Test data and labels must be the same size");
##         }
##         std::iota(trainingInd.begin(), trainingInd.end(), 0);
## };
##
```

```
## // Destructor
## template <typename DT, typename LT>
## Optimizer<DT, LT>::~Optimizer() {};
##
##
## // Optimizer member functions
## // ------------------------------------------------------------------------
##
## // Gets a batch of data and labels from the training data
## template <typename DT, typename LT>
## std::pair<std::vector<DT>, std::vector<LT>> Optimizer<DT, LT>::getTrainBatch(const int& batchNum) co
##     if (batchNum >= numBatches) {
##         throw std::invalid_argument("Batch number must be less than the number of batches");
##     }
##     std::vector<DT> batchData(batchSize);
##     std::vector<LT> batchLabels(batchSize);
##     for (int i = 0; i < batchSize; i++) {
##         batchData[i] = trainingData[trainingInd[batchNum * batchSize + i]];
##         batchLabels[i] = trainingLabels[trainingInd[batchNum * batchSize + i]];
##     }
##     return std::make_pair(batchData, batchLabels);
## };
##
## // Gets all the training data
## template <typename DT, typename LT>
## std::pair<std::vector<DT>, std::vector<LT>> Optimizer<DT, LT>::getTrainData() const {
##     return std::make_pair(trainingData, trainingLabels);
## };
##
## // Gets all validation data and labels
## template <typename DT, typename LT>
## std::pair<std::vector<DT>, std::vector<LT>> Optimizer<DT, LT>::getValidation() const {
##     return std::make_pair(validationData, validationLabels);
## };
##
## // Gets a testing data and labels
## template <typename DT, typename LT>
## std::pair<std::vector<DT>, std::vector<LT>> Optimizer<DT, LT>::getTest() const {
##     return std::make_pair(testData, testLabels);
## };
##
## // Shuffles the training data and labels
## template <typename DT, typename LT>
## void Optimizer<DT, LT>::shuffleTrainData() {
##     std::random_device rd;  // Obtain a random seed from the hardware
##     std::mt19937 eng(rd()); // Seed the random number engine
##     std::shuffle(trainingInd.begin(), trainingInd.end(), eng); // Shuffle the vector of indices
## };
##
## // Performs one epochs worth of training
## template <typename DT, typename LT>
## std::vector<double> Optimizer<DT, LT>::singleTrain() {
##     // Shuffle the indices we use to access training data
##     shuffleTrainData();
```

```
##     // Container for the average training loss for each batch
##     std::vector<double> avg_train_loss(numBatches);
##     // Perform one epoch of training
##     for(int i = 0 ; i < numBatches ; ++i){
##         std::cout << "Batch " << i << std::endl;
##         // Get a batch of data and labels
##         std::pair<std::vector<DT>, std::vector<LT>> batch = getTrainBatch(i);
##
##         // Carry out mini-batch SGD on the batch
##         avg_train_loss[i] = nn.backpropagation(batch.first, batch.second, learningRate);
##     }
##     return avg_train_loss;
## };
##
## // Performs a single testing step, ie. forward pass and loss calculation for whole test set
## template <typename DT, typename LT>
## double Optimizer<DT, LT>::singleTest(){
##     double loss = 0.0;
##     for(int i = 0 ; i < testData.size() ; ++i){
##         // Carry out forward pass on the batch
##         std::vector<double> out = nn.forwardPass(testData[i]);
##         loss += computeMSE(out, testLabels[i]);
##     }
##     std::cout << "Test loss: " << loss << std::endl;
##     return loss / testData.size();
## };
##
## // Iteratively conducts training step for every batch and for number of specified epochs
## template <typename DT, typename LT>
## void Optimizer<DT, LT>::train(){
##     // Container for the average training loss for each batch
##     std::vector<std::vector<double>> avg_train_loss(numEpochs, std::vector<double>(numBatches));
##     // Container for the average test loss for each epoch
##     std::vector<double> avg_test_loss(numEpochs);
##     // Perform the specified number of epochs
##     for(int i = 0 ; i < numEpochs ; ++i){
##         std::cout << "Epoch " << i << std::endl;
##         // Perform one epoch of training
##         avg_train_loss[i] = singleTrain();
##         // Perform one epoch of testing
##         avg_test_loss[i] = singleTest();
##     }
##     // Print the average training loss for each batch
##     std::cout << "Average training loss for each batch:" << std::endl;
##     printMatrix(avg_train_loss);
##     // Print the average test loss for each epoch
##     std::cout << "Average test loss for each epoch:" << std::endl;
##     printVector(avg_test_loss);
## };
##
## // Returns the neural network
## template <typename DT, typename LT>
## NeuralNetwork Optimizer<DT, LT>::getNn() const {
##     return nn;
```

```
## };
##
## // Returns the training indices
## template <typename DT, typename LT>
## std::vector<int> Optimizer<DT, LT>::getTrainingInd() const {
##     return trainingInd;
## };
```

Finally I will print out the unit tests:

```
cat neural_network/Unit_Tests.cpp
```

```
## #include "gtest/gtest.h"
## #include "Print_Utils.h"
## #include "Neural_Net.h"
## #include "Optimizer.h"
##
## // Test randomDouble function with positive and negative values
## TEST(RandomDoubleTest, PositiveValues) {
##     double result = randomDouble(1.0, 2.0);
##     EXPECT_TRUE(result >= 1.0 && result <= 2.0);
## }
## TEST(RandomDoubleTest, NegativeValues) {
##     double result = randomDouble(-2.0, -1.0);
##     EXPECT_TRUE(result >= -2.0 && result <= -1.0);
## }
##
##  // Test initializeWeights function
## TEST(InitializeWeightsTest, DifferentSizes) {
##     std::vector<std::vector<double>> weights1 = initializeWeights(2, 3, 0.0, 1.0);
##     EXPECT_EQ(weights1.size(), 2);
##     EXPECT_EQ(weights1[0].size(), 3);
##     std::vector<std::vector<double>> weights2 = initializeWeights(4, 5, -1.0, 0.0);
##     EXPECT_EQ(weights2.size(), 4);
##     EXPECT_EQ(weights2[0].size(), 5);
## }
##
## // Test initializeBiases function
## TEST(initializeBiasesTest, PositiveValues){
##     std::vector<double> biases = initializeBiases(5, 1.0, 2.0);
##     EXPECT_EQ(biases.size(), 5);
##     for (double bias : biases) {
##         EXPECT_TRUE(bias >= 1.0 && bias <= 2.0);
##     }
## }
## TEST(initializeBiasesTest, ZeroValues){
##     EXPECT_THROW(initializeBiases(0, 1.0, 2.0), std::invalid_argument);
## }
## TEST(initializeBiasesTest, InvalidValues){
##     EXPECT_THROW(initializeBiases(5, 2.0, 1.0), std::invalid_argument);
## }
##
## // Test ReLU class
## TEST(ReLUClassTest, Activate) {
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
```

```
##     double result = relu->activate(1.0);
##     EXPECT_DOUBLE_EQ(result, 1.0);
## }
## TEST(ReLUClassTest, Derivative) {
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
##     double result = relu->derivative(1.0);
##     EXPECT_DOUBLE_EQ(result, 1.0);
## }
## TEST(ReLUClassTest, GetName) {
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
##     std::string result = relu->get_name();
##     EXPECT_EQ(result, "ReLU");
## }
##
##  // Test Layer class
## TEST(LayerTest, Constructor) {
##     std::vector<std::vector<double>> weights = {{1.0, 2.0}, {3.0, 4.0}};
##     std::vector<double> biases = {1.0, 2.0};
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
##     Layer layer(weights, biases, relu);
##     std::vector<std::vector<double>> result = layer.get_weights();
##     EXPECT_EQ(result.size(), 2);
##     EXPECT_EQ(result[0].size(), 2);
##     EXPECT_DOUBLE_EQ(result[0][0], 1.0);
##     EXPECT_DOUBLE_EQ(result[0][1], 2.0);
##     EXPECT_DOUBLE_EQ(result[1][0], 3.0);
##     EXPECT_DOUBLE_EQ(result[1][1], 4.0);
## }
## TEST(LayerTest, ConstructorWithValidInputs) {
##     std::vector<std::vector<double>> weights = {{1,2,3},{4,5,6}};
##     std::vector<double> biases = {1,2};
##     auto actFun = std::make_shared<ReLU>();
##     Layer layer(weights, biases, actFun);
##     ASSERT_EQ(layer.get_weights(), weights);
##     ASSERT_EQ(layer.get_biases(), biases);
##     ASSERT_EQ(layer.get_actFun(), actFun);
## }
## TEST(LayerTest, LayerPass) {
##     std::vector<std::vector<double>> weights = {{1.0, 2.0}, {3.0, 4.0}};
##     std::vector<double> biases = {1.0, 2.0};
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
##     Layer layer(weights, biases, relu);
##     std::vector<double> inputs = {0.5, -0.5};
##     std::vector<double> result = layer.layer_pass(inputs);
##     EXPECT_EQ(result.size(), 2);
##     EXPECT_DOUBLE_EQ(result[0], 0.5);
##     EXPECT_DOUBLE_EQ(result[1], 1.5);
## }
##
## TEST(LayerTest, SetWeightsPositiveTestCase) {
##     std::vector<std::vector<double>> weights = {{1.0, 2.0}, {3.0, 4.0}};
##     std::vector<double> biases = {1.0, 2.0};
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
##     Layer layer(weights, biases, relu);
```

```
##     std::vector<std::vector<double>> newWeights {{1, 2}, {3, 4}};
##     layer.set_weights(newWeights);
##     // Check that the weights were set correctly
##     EXPECT_EQ(layer.get_weights(), newWeights);
## }
## TEST(LayerTest, SetWeightsNegativeTestCase) {
##     std::vector<std::vector<double>> weights = {{1.0, 2.0}, {3.0, 4.0}};
##     std::vector<double> biases = {1.0, 2.0};
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
##     Layer layer(weights, biases, relu);
##     std::vector<std::vector<double>> newWeights {{1, 2}, {3, 4}};
##     // Try to set the weights with a matrix of different dimensions
##     std::vector<std::vector<double>> wrongDimensions {{1, 2}, {3, 4}, {5, 6}};
##     EXPECT_THROW(layer.set_weights(wrongDimensions), std::invalid_argument);
##     // Check that the weights were not set to the wrong dimensions
##     EXPECT_NE(layer.get_weights(), wrongDimensions);
## }
##
## TEST(LayerTest, GetBiases) {
##     std::vector<std::vector<double>> weights = {{1.0, 2.0}, {3.0, 4.0}};
##     std::vector<double> biases = {1.2, 3.4};
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
##     Layer layer(weights, biases, relu);
##     std::vector<double> expected_biases = {1.2, 3.4};
##     layer.set_biases(expected_biases);
##     std::vector<double> actual_biases = layer.get_biases();
##     ASSERT_EQ(actual_biases, expected_biases);
## }
##
## TEST(LayerTest, GetBiasesEmpty) {
##     std::vector<std::vector<double>> weights = {{1.0, 2.0}, {3.0, 4.0}};
##     std::vector<double> biases = {1.0, 2.0};
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
##     Layer layer(weights, biases, relu);
##     std::vector<double> expected_biases = {};
##      std::vector<double> actual_biases = layer.get_biases();
##      ASSERT_NE(actual_biases, expected_biases);
## }
## TEST(LayerTest, SetBiasesPositive) {
##     std::vector<std::vector<double>> weights = {{1.0, 2.0}, {3.0, 4.0}};
##     std::vector<double> biases = {1.0, 2.0};
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
##     Layer layer(weights, biases, relu);
##     std::vector<double> newBiases = {-0.5, 0.5};
##     layer.set_biases(newBiases);
##     EXPECT_EQ(layer.get_biases(), newBiases);
## }
##  TEST(LayerTest, SetBiasesNegative) {
##     std::vector<std::vector<double>> weights = {{1.0, 2.0}, {3.0, 4.0}};
##     std::vector<double> biases = {1.0, 2.0};
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
##     Layer layer(weights, biases, relu);
##     std::vector<double> newBiases = {-0.5, 0.0, 2.0};  // wrong size
##     EXPECT_THROW(layer.set_biases(newBiases), std::invalid_argument);
```

```
##     EXPECT_NE(layer.get_biases(), newBiases);
## }
## TEST(LayerTest, ActivationDerivativeTest) {
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
##     Layer layer({{1.0, 2.0}, {3.0, 4.0}}, {1.0, 1.0}, relu);
##     std::vector<double> input = {0.5, -0.5};
##     layer.layer_pass(input);
##     std::vector<double> expectedDerivative = {1.0, 1.0};
##     EXPECT_EQ(layer.get_activation_derivative(), expectedDerivative);
## }
##
##
##  // Test NeuralNetwork class
## TEST(NeuralNetworkTest, ForwardPass) {
##     std::vector<std::vector<double>> weights1 = {{1.0,-1.0}, {-1.0, 2.0}};
##     std::vector<double> biases1 = {1.0, 1.0};
##     std::vector<std::vector<double>> weights2 = {{1.0, -1.0}, {-1.0, 2.0}};
##     std::vector<double> biases2 = {1.0, 0.5};
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
##     Layer layer1(weights1, biases1, relu);
##     Layer layer2(weights2, biases2, relu);
##     std::vector<Layer> layers = {layer1, layer2};
##     NeuralNetwork nn(layers);
##     std::vector<double> inputs = {0.5, -0.5};
##     std::vector<double> result = nn.forwardPass(inputs);
##     EXPECT_EQ(result.size(), 2);
##     EXPECT_DOUBLE_EQ(result[0], 3);
##     EXPECT_DOUBLE_EQ(result[1], 0);
## }
##
##  // Negative test cases
## TEST(InitializeWeightsTest, MinBiggerThanMax) {
##   EXPECT_THROW(initializeWeights(5, 5, 5.0, 2.0), std::invalid_argument);
## }
## TEST(LayerTest, InvalidWeightsSize) {
##     std::vector<std::vector<double>> weights = {{1.0}, {2.0}};
##     std::vector<double> biases = {1.0, 2.0};
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
##     EXPECT_THROW(Layer layer(weights, biases, relu), std::invalid_argument);
## }
## TEST(LayerTest, InvalidInputsSize) {
##     std::vector<std::vector<double>> weights = {{1.0, 2.0}, {3.0, 4.0}};
##     std::vector<double> biases = {1.0, 2.0};
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
##     Layer layer(weights, biases, relu);
##     std::vector<double> inputs = {0.5};
##     EXPECT_THROW(layer.layer_pass(inputs), std::invalid_argument);
## }
## TEST(NeuralNetworkTest, InvalidLayersSize) {
##     std::vector<std::vector<double>> weights1 = {{1.0, 2.0}, {3.0, 4.0}};
##     std::vector<double> biases1 = {1.0, 2.0};
##     std::vector<std::vector<double>> weights2 = {{5.0, 6.0, 7.0}, {8.0, 9.0, 10.0}};
##     std::vector<double> biases2 = {1.0, 2.0};
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
```

```
##     Layer layer1(weights1, biases1, relu);
##     Layer layer2(weights2, biases2, relu);
##     std::vector<Layer> layers = {layer1, layer2};
##     EXPECT_THROW(NeuralNetwork nn(layers), std::invalid_argument);
## }
##
## // Test computeLossGradient function
##  TEST(NeuralNetworkTest, ComputeLossPositive) {
##     std::vector<std::vector<double>> weights1 = {{1.0,-1.0}, {-1.0, 2.0}};
##     std::vector<double> biases1 = {1.0, 1.0};
##     std::vector<std::vector<double>> weights2 = {{1.0, -1.0}, {-1.0, 2.0}};
##     std::vector<double> biases2 = {1.0, 0.5};
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
##     Layer layer1(weights1, biases1, relu);
##     Layer layer2(weights2, biases2, relu);
##     std::vector<Layer> layers = {layer1, layer2};
##     NeuralNetwork net(layers);
##     std::vector<double> input = {0.5, -0.5};
##     std::vector<double> output = net.forwardPass(input);
##     std::vector<double> target = {2.0, 1.0};
##     std::vector<double> expectedLossGradient = {2.0, -2.0};
##     std::vector<double> actualLossGradient = net.computeLossGradient(output, target);
##     EXPECT_EQ(expectedLossGradient, actualLossGradient);
## }
## TEST(NeuralNetworkTest, ComputeLossNegative) {
##     std::vector<std::vector<double>> weights1 = {{1.0, 2.0}, {3.0, 4.0}};
##     std::vector<double> biases1 = {1.0, 2.0};
##     std::vector<std::vector<double>> weights2 = {{5.0, 6.0}, {8.0, 9.0}};
##     std::vector<double> biases2 = {1.0, 2.0};
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
##     Layer layer1(weights1, biases1, relu);
##     Layer layer2(weights2, biases2, relu);
##     std::vector<Layer> layers = {layer1, layer2};
##     NeuralNetwork net(layers);
##     std::vector<double> input = {1.0, 1.0};
##     std::vector<double> output = net.forwardPass(input);
##     std::vector<double> target = {1.0, 2.0, 3.0}; // Different size than output
##     EXPECT_THROW(net.computeLossGradient(output, target), std::invalid_argument);
## }
##
## //  Test computeWeightGradients function
## TEST(NeuralNetworkTest, ComputeWeightGradients_PositiveTest) {
##     // Test delta input
##     std::vector<double> delta = {0.1, 0.2, 0.3};
##      // Test layer weights
##     std::vector<std::vector<double>> weights = {{0.5, 0.3, 0.1}, {0.2, 0.4, 0.6}, {0.8, 0.9, 0.3}};
##     std::vector<double> biases = {10.0, 0.0, 2.0};
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
##     Layer testLayer(weights, biases, relu);
##     std::vector<Layer> layers = {testLayer};
##     NeuralNetwork net(layers);
##     // Test expected output
##     std::vector<std::vector<double>> expectedGradients = {{0.05, 0.03, 0.01}, {0.04, 0.08, 0.12}, {0
##     // Compute weight gradients
```

```
##     std::vector<std::vector<double>> actualGradients = net.computeWeightGradients(delta, testLayer);
##     // Check that actual gradients match expected gradients to within 1e-6 tolerance
##     for (size_t i = 0; i < expectedGradients.size(); ++i) {
##         for (size_t j = 0; j < expectedGradients[0].size(); ++j) {
##             EXPECT_NEAR(actualGradients[i][j], expectedGradients[i][j], 1e-6);
##         }
##     }
## }
## TEST(NeuralNetworkTest, ComputeWeightGradients_NegativeTest) {
##     // Test delta input with wrong size
##     std::vector<double> delta = {0.1, 0.2, 0.3, 0.4};
##     // Test layer weights
##     std::vector<std::vector<double>> weights = {{0.5, 0.3, 0.1}, {0.2, 0.4, 0.6}, {0.8, 0.9, 0.3}};
##     std::vector<double> biases = {0.0, 0.0, 2.0};
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
##     Layer testLayer(weights, biases, relu);
##     std::vector<Layer> layers = {testLayer};
##     NeuralNetwork net(layers);
##     // Check that an invalid argument exception is thrown
##     EXPECT_THROW(net.computeWeightGradients(delta, testLayer), std::invalid_argument);
## }
## //  Test computeBiasGradients function
## TEST(NeuralNetworkTest, ComputeBiasGradients) {
##     std::vector<std::vector<double>> weights = {{0.5, 0.3, 0.1}, {0.2, 0.4, 0.6}, {0.8, 0.9, 0.3}};
##     std::vector<double> biases = {5.0, 0.0, 1.0};
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
##     Layer testLayer(weights, biases, relu);
##     std::vector<Layer> layers = {testLayer};
##     NeuralNetwork net(layers);
##     std::vector<double> delta = {0.1, -0.2, 0.3};
##     std::vector<double> expected_output = {0.1, -0.2, 0.3};
##     std::vector<double> output = net.computeBiasGradients(delta);
##     // Check that the output is the expected size
##     ASSERT_EQ(output.size(), expected_output.size());
##     // Check that each element of the output is equal to the expected output
##     for (size_t i = 0; i < output.size(); ++i) {
##         EXPECT_EQ(output[i], expected_output[i]);
##     }
## }
## // Test the updateWeightsAndBiases function
## TEST(NeuralNetworkTest, UpdateWeightsAndBiases_ValidInput) {
##     std::vector<std::vector<double>> weights = {{0.1, 0.2, 0.3}, {0.4, 0.5, 0.6}, {0.7, 0.8, 0.9}};
##     std::vector<double> biases = {1.0, 2.0, 3.0};
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
##     Layer testLayer(weights, biases, relu);
##     std::vector<Layer> layers = {testLayer};
##     NeuralNetwork net(layers);
##     std::vector<std::vector<double>> weightGradients = {{2, 4, 6}, {8, 10, 12}, {14, 16, 18}};
##     std::vector<double> biasGradients = {3, 6, 9};
##     double learningRate = 0.1;
##     net.updateWeightsAndBiases(testLayer, weightGradients, biasGradients, learningRate);
##     std::vector<std::vector<double>> expected_weights = {{-0.1, -0.2, -0.3}, {-0.4, -0.5, -0.6}, {-0
##     std::vector<double> expected_biases = {0.7, 1.4, 2.1};
##     for (size_t i = 0; i < expected_weights.size(); ++i) {
```

```
##         for (size_t j = 0; j < expected_weights[0].size(); ++j) {
##             EXPECT_NEAR(testLayer.get_weights()[i][j], expected_weights[i][j], 1e-6);
##         }
##     }
##     EXPECT_EQ(testLayer.get_biases(), expected_biases);
## }
## TEST(NeuralNetworkTest, updateWeightsAndBiasesNegativeTest) {
##     std::vector<std::vector<double>> weights = {{0.5, 0.3, 0.1}, {0.2, 0.4, 0.6}, {0.8, 0.9, 0.3}};
##     std::vector<double> biases = {5.0, 0.0, 1.0};
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
##     Layer testLayer(weights, biases, relu);
##     std::vector<Layer> layers = {testLayer};
##     NeuralNetwork net(layers);
##     // Set up test input
##     std::vector<double> input{ 1.0, 2.0, 3.0 };
##     // Set up test gradients with incorrect dimensions
##     std::vector<std::vector<double>> weightGradients{ {0.1, 0.2, 0.3}, {0.4, 0.5, 0.6} };
##     std::vector<double> biasGradients{ 0.1 };
##     // Call updateWeightsAndBiases function with incorrect dimensions of gradients
##     EXPECT_THROW(net.updateWeightsAndBiases(testLayer, weightGradients, biasGradients, 0.1), std::in
## }
## // Test the computeDelta function
## TEST(NeuralNetworkTest, ComputeDeltaTest) {
##     std::vector<std::vector<double>> weights = {{0.5, 0.3}, {0.2, 0.4}};
##     std::vector<double> biases = {1.0, -1.0};
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
##     Layer testLayer(weights, biases, relu);
##     testLayer.layer_pass({1.0, 2.0});
##     std::vector<Layer> layers = {testLayer};
##     NeuralNetwork net(layers);
##     std::vector<double> delta = {1.0, 2.0};
##     std::vector<double> previousDelta = net.computeDelta(delta, testLayer);
##     ASSERT_EQ(previousDelta.size(), 2);
##     EXPECT_DOUBLE_EQ(previousDelta[0], 0.9);
##     EXPECT_DOUBLE_EQ(previousDelta[1], 0);
## }
## TEST(NeuralNetworkTest, ComputeDeltaNegativeTest) {
##     std::vector<std::vector<double>> weights = {{0.5, 0.3, 0.1}, {0.2, 0.4, 0.6}, {0.8, 0.9, 0.3}};
##     std::vector<double> biases = {5.0, 0.0, 1.0};
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
##     Layer testLayer(weights, biases, relu);
##     std::vector<Layer> layers = {testLayer};
##     NeuralNetwork net(layers);
##     std::vector<double> delta = {1, 0}; // Delta size does not match the number of output neurons
##     ASSERT_THROW(net.computeDelta(delta, testLayer), std::invalid_argument);
## }
##
##
##
## // Tests for Optimizer.h
## // ------------------------------------------------------------------------------------------
## // Test for getTrainBatch
## TEST(OptimizerTest, getTrainBatchTest) {
##     std::vector<std::vector<double>> trainData = {{1, 2}, {4, 5}, {7, 8}};
```

```
##     std::vector<double> trainLabels = {3, 9, 15};
##
##     std::vector<std::vector<double>> valData = {{-3, 4}, {2, 5}, {6, 3}};
##     std::vector<double> valLabels = {1, 7, 9};
##
##     std::vector<std::vector<double>> testData = {{10, 11}, {13, 14}};
##     std::vector<double> testLabels = {21, 27};
##     int batchSize = 2;
##
##     std::vector<std::vector<double>> expectedTrainBatchData = {{1, 2}, {4, 5}};
##     std::vector<double> expectedTrainBatchLabels = {3, 9};
##
##     NeuralNetwork nn({Layer(2, 2, std::make_shared<ReLU>())});
##     Optimizer<std::vector<double>, double> optimizer(nn, trainData, trainLabels, testData, testLabels
##     auto result = optimizer.getTrainBatch(0);
##     EXPECT_EQ(result.first, expectedTrainBatchData);
##     EXPECT_EQ(result.second, expectedTrainBatchLabels);
## }
## // Test for getValidation
## TEST(OptimizerTest, getValidationTest) {
##     std::vector<std::vector<double>> trainData = {{1, 2}, {4, 5}, {7, 8}};
##     std::vector<double> trainLabels = {3, 9, 15};
##
##     std::vector<std::vector<double>> valData = {{-3, 4}, {2, 5}, {6, 3}};
##     std::vector<double> valLabels = {1, 7, 9};
##
##     std::vector<std::vector<double>> testData = {{10, 11}, {13, 14}};
##     std::vector<double> testLabels = {21, 27};
##     int batchSize = 2;
##
##     std::vector<std::vector<double>> expectedValData = {{-3, 4}, {2, 5}, {6, 3}};
##     std::vector<double> expectedValLabels = {1, 7, 9};
##
##     NeuralNetwork nn({Layer(2, 2, std::make_shared<ReLU>())});
##     Optimizer<std::vector<double>, double> optimizer(nn, trainData, trainLabels, testData, testLabels
##     auto result = optimizer.getValidation();
##     EXPECT_EQ(result.first, expectedValData);
##     EXPECT_EQ(result.second, expectedValLabels);
## }
## // Test for getTest
## TEST(OptimizerTest, getTestTest) {
##     std::vector<std::vector<double>> trainData = {{1, 2}, {4, 5}, {7, 8}};
##     std::vector<double> trainLabels = {3, 9, 15};
##
##     std::vector<std::vector<double>> valData = {{-3, 4}, {2, 5}, {6, 3}};
##     std::vector<double> valLabels = {1, 7, 9};
##
##     std::vector<std::vector<double>> testData = {{10, 11}, {13, 14}};
##     std::vector<double> testLabels = {21, 27};
##     int batchSize = 2;
##
##     std::vector<std::vector<double>> expectedTestData = {{10, 11}, {13, 14}};
##     std::vector<double> expectedTestLabels = {21, 27};
##
```

```
##     NeuralNetwork nn({Layer(2, 2, std::make_shared<ReLU>())});
##     Optimizer<std::vector<double>, double> optimizer(nn, trainData, trainLabels, testData, testLabels
##     auto result = optimizer.getTest();
##     EXPECT_EQ(result.first, expectedTestData);
##     EXPECT_EQ(result.second, expectedTestLabels);
## }
## // Test for shuffleTrainData
## TEST(OptimizerTest, shuffleTrainData) {
##     std::vector<std::vector<double>> trainData = {{1, 2}, {4, 5}, {7, 8}};
##     std::vector<double> trainLabels = {3, 9, 15};
##
##     std::vector<std::vector<double>> testData = {{10, 11}, {13, 14}};
##     std::vector<double> testLabels = {21, 27};
##     int batchSize = 2;
##     NeuralNetwork nn({Layer(2, 2, std::make_shared<ReLU>())});
##     Optimizer<std::vector<double>, double> optimizer(nn, trainData, trainLabels, testData, testLabels
##
##     std::vector<int> preShuffledInd = optimizer.getTrainingInd();
##     optimizer.shuffleTrainData();
##     std::vector<int> shuffledInd = optimizer.getTrainingInd();
##     EXPECT_NE(shuffledInd, preShuffledInd);
## }
##
## // Test for singleTrain
## TEST(OptimizerTest, singleTrainTest) {
##     std::vector<std::vector<double>> trainData = {{0, 0}, {0, 0}, {0, 0}};
##     std::vector<std::vector<double>> trainLabels = {{1,0}, {0,1}, {1, 0}};
##
##     std::vector<std::vector<double>> testData = {{10, 11}, {13, 14}};
##     std::vector<std::vector<double>> testLabels = {{1, 0}, {0, 1}};
##     int batchSize = 2;
##
##     std::vector<std::vector<double>> weights = {{1.0,-1.0}, {-1.0, 2.0}};
##     std::vector<double> biases = {0.0, 0.0};
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
##     Layer layer(weights, biases, relu);
##     std::vector<Layer> layers = {layer};
##     NeuralNetwork nn(layers);
##
##     Optimizer<std::vector<double>, std::vector<double>> optimizer(nn, trainData, trainLabels, testDa
##     std::vector<double> loss = optimizer.singleTrain();
##     EXPECT_EQ(loss.size(), 1);
##     EXPECT_NEAR(loss[0], 0.5, 1e-6);
## }
##
## // Test for singleTest
## TEST(OptimizerTest, singleTestTest) {
##     std::vector<std::vector<double>> trainData = {{0, 0}, {0, 0}, {0, 0}};
##     std::vector<std::vector<double>> trainLabels = {{1,0}, {0,1}, {1, 0}};
##
##     std::vector<std::vector<double>> testData = {{0, 0}, {0, 0}};
##     std::vector<std::vector<double>> testLabels = {{1, 0}, {0, 1}};
##     int batchSize = 2;
##
```

```
##     std::vector<std::vector<double>> weights = {{1.0,-1.0}, {-1.0, 2.0}};
##     std::vector<double> biases = {0.0, 0.0};
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
##     Layer layer(weights, biases, relu);
##     std::vector<Layer> layers = {layer};
##     NeuralNetwork nn(layers);
##
##     Optimizer<std::vector<double>, std::vector<double>> optimizer(nn, trainData, trainLabels, testDa
##     double loss = optimizer.singleTest();
##     EXPECT_NEAR(loss, 0.5, 1e-6);
## }
##
## // Test for train
## TEST(OptimizerTest, trainTest) {
##     std::vector<std::vector<double>> trainData = {{0, 0}, {0, 0}, {0, 0}};
##     std::vector<std::vector<double>> trainLabels = {{1,0}, {0,1}, {1, 0}};
##
##     std::vector<std::vector<double>> testData = {{0, 0}, {0, 0}};
##     std::vector<std::vector<double>> testLabels = {{1, 0}, {0, 1}};
##     int batchSize = 2;
##
##     std::vector<std::vector<double>> weights = {{1.0,-1.0}, {-1.0, 2.0}};
##     std::vector<double> biases = {0.0, 0.0};
##     std::shared_ptr<ActivationFunction> relu = std::make_shared<ReLU>();
##     Layer layer(weights, biases, relu);
##     std::vector<Layer> layers = {layer};
##     NeuralNetwork nn(layers);
##
##     Optimizer<std::vector<double>, std::vector<double>> optimizer(nn, trainData, trainLabels, testDa
##     optimizer.train();
##     // EXPECT_EQ(loss.size(), 1);
##     // EXPECT_NEAR(loss[0], 0.5, 1e-6);
## }
##
## int main(int argc, char **argv) {
##     testing::InitGoogleTest(&argc, argv);
##     return RUN_ALL_TESTS();
## }
```