## Member Summary

| Type | Name | Description |
|---|---|---|
| `double` | `learning_rate` | Multiplier for updating weights and biases |
| `int` | `batch_size` | Size of mini batches to take from test data |
| `int` | `epochs` | Number of times to "train" the network // consider changing name to more descriptive one |
| `vector<int>` | `layer_sizes` | Array containing sizes of each layer |
| `int` | `num_Layers` | Number of layers in the network |
| `int` | `num_correct` | Records the number of correct outputs generated by the network |
| `vector<matrix>` | `weights` | Vector of matrices containing all the weights for the network |
| `vector<matrix>` | `biases` | Vector of matrices containing all the biases for the network |
| `vector<matrix>` | `activations` | Vector of matrices containing all the activation values of the network |
| `vector<matrix>` | `weighted_inputs` | Vector of matrices containing all the weighted sums of the network |
| `vector<matrix>` | `errors` | Vector of matrices containing all errors at each node of the network |
| `vector<matrix>` | `sum_nabla_b` | Vector of matrices containing the sum of the dC/db at each node of the network |
| `vector<matrix>` | `sum_nabla_w` | Vector of matrices containing the sum of the dC/dw at each node of the network |
| `vector<matrix>` | `expected_values` | Vector of matrices containing the expected values for each training data input |
| `vector<int>` | `test_data_indices` | Array of all indices corresponding to the test data |
| `vector<int>` | `min_batch_indices` | Array of indices of size batch_size corresponding to the test data |
| `string` | `training_data_filename` | Name of file of test data inputs |
| `string` | `expected_values_filename` | Name of file of expected values that correspond to the test data inputs |
| `ifstream` | `training_data_infile` | Ifstream object to read in from the training_data_file |
| `ifstream` | `expected_values_infile` | Ifstream object to read in from the expected_values_file |

## Constructor Summary

| Constructor | Description |
|---|---|
| `Network (istream& = cin)` | Constructs an untrained Network object that represents the data read from an input stream |
| `Network (const string& , const string&)` | Reconstructs a [trained] network and classifies it |
| `Network (const string&)` | Reconstructs a trained nework and trains it again |
| `~Network ()` | Deallocates the Network object's dynamic memory and closes all files |

## Method Summary

| Return type | Name | Description |
|---|---|---|
| `void` | `readInit (const string&)` | Reads the required hyperparameters of the class from a file given as a parameter |
| `void` | `readInit ()` | Reads the required hyperparameters from the console `cin` |
| `void` | `forwardPropagation (int)` | Sets all activation values and weighted inputs for a single test data input |
| `void` | `backPropagation (int)` | Backpropagates through network to compute error at each node |
| `void` | `SGD ()` | Stochastic Gradient Descent: performs forward and back propagation once for each input in the mini batch then updates the weights and biases accordingly |
| `void` | `train ()` | Trains network by repeatedly performing SGD on randomized mini batches of the test data for as many epochs as specified |
| `void` | `update ()` | Updates weights and biases based on data from SGD |
| `void` | `classify ()` | Classifies the data file provided by the user and displays the efficiency based on the classified data |
| `void` | `randomizeMatrix (matrix<double> , double * distribution())` | Assigns Gaussian normally distributed set of random numbers for each element in a matrix |
| `bool` | `createNetworkFile ()` | Writes a file to store the network |
| `matrix&` | `hadamardProduct (const matrix& , const matrix&)` | Performs the Hadamard Product operation on any two given matrices |
| `void` | `shuffleDataIndices(vector<T>)` | Shuffles the data indices in a given vector |

## Member Details

### learning_rate

```
double learning_rate;
```

Positive multiplier for updating weights and biases.
Generally, the learning rate is less than 1

### batch_size

```
int batch_size;
```

Size of the small samples of randomly chosen training inputs from the test data file. Provided, the sample size should be large enough to speedup the learning process of the network.

### epochs

```
int epochs;
```

A hyperparameter that denotes the number of times the samples of finite size needs to be chosen randomly, such that the training inputs are exhausted. This number is usually provided by the user rather than a randomly selected value by the learning algorithm.

### layer_sizes

```
vector<int> layer_sizes;
```

Records the size of every layer in the network, once given a source by the user. Every required matrix in the class gets the size using this parameter.

### num_Layers

```
int num_Layers;
```

Stores information about the depth of the network. This parameter, particularly, helps in randomly accessing any particular layer in the network.

### num_correct

```
int num_correct;
```

This records the number of correct outputs generated by the network after comparing them to the expected values. It helps in getting the efficiency of the network after every epoch.

### weights

```
vector<matrix> weights;
```

The weights of all the layers are stored in a vector of matrices each of size (j x k), where k is the layer size just before the layer with j as its layer size. Initially, the weights are randomly assigned values using the Guassian normal distribution. Though, the weights get updated once the network starts learning.

### biases

```
vector<matrix> biases;
```

The biases of all the layers in a network are stored in a vector of matrices each of size (j x 1), where j is the layer size of that layer. Initially, the biases are randomly assigned values using the Gaussian normal distribution. However, the biases get updated as the network starts learning.

## activations

```
vector<matrix> activations;
```
The activations of a particular layer in a network are stored in a matrix of size (j x 1), where j is the layer size of that layer. Activation values are calculated by the activation function declared outside the class. It requires the weighted sums to perform the calculation. However, for every training input, the first layer of activations will be the training input data extracted from the file.

## weighted_inputs

```
vector<matrix> weighted_inputs;
```
The weighted sums of all layers in a network are stored in a vector of matrices each of size (j x 1), where j is the layer size of that layer. The weighted sum at any particular layer can be given by the dot product, of the weight matrix for that layer and the values of the activations layer just before that layer, with the bias matrix for that layer added to the dot product.

**weighted_inputs at jth layer = ((weights at j) . (activations at j - 1)) + (biases at j)**

## errors

```
vector<matrix> errors;
```
The errors for all the layers in a network are stored in a vector of matrices each of size (j x 1), where j is the layer size of that layer. The errors for the ouput layer will be calculated first using the expected values and the derivative of the activation function. Then, the errors will be back propagated to all the hidden layers.

## sum_nabla_b

```
vector<matrix> sum_nabla_b;
```
The sum of the cost partials with respect to biases for all the layers in a network are stored in a vector of matrices each of size (j x 1), where j is the layer size of that layer. This stores the sum of the dC/db values for all the layers in the network. The cost partial, dC/db for a specific layer is directly equal to the error in that specific layer.

## sum_nabla_w

```
vector<matrix> sum_nabla_w;
```
The sum of the cost partials with respect to weights for all the layers in a network are stored in a vector of matrices each of size (j x k), where k is the layer size just before the layer with j as its layer size. This stores the sum of the dC/dw values for all the layers in the network. For any *ith* layer in the network,

**dC/dw = Hadamard Product of [((errors at *ith* layer) . ((activations at *(i-1)th* layer) $^T$ ))] and [Sigmoid Prime of weighted_inputs at *ith* layer]**

## Expected_Values

```
vector<matrix> Expected_values;
```
The Expected_Values is a vector of a matrix of size (L x 1), where L is the size of the ouput layer in the network. The expected values for every training data input are extracted from the file and stored here to compare the final output after every forward propagation with these expected values.

## test_data_indices

```
vector<int> test_data_indices;
```
A vector of the size same as that of the number of training data inputs. Randomization after every epoch is performed on the test data indices before passing them to any other function.

## mini_batch_indices

```
vector<int> mini_batch_indices;
```

A vector of the size same as the batch size. This parameter stores in the indices after the randomization and it is made sure that the different mini batches get unique indices from the test data.

## test_data_file

```
string test_data_file;
```

The name of the file containing the test data is the file from which the activation values at the first layer is extracted. This also gives the size to the test_data_indices.

## expected_values_file

```
string expected_values_file;
```

The expected_values_file gives the expected values for each training data input in the test_data_file. It helps in getting the values of errors and helps the network to learn faster.

## Constructor Details

### Default Constructor

**Syntax:**

```
        Network::Network (istream& in = cin);
```

**Parameters:**

It takes one parameter, which is the input stream. Although, this input stream is by default set to **cin**.

**Description:**

This constructor assigns values to all the data members, based on the values of some data members entered by the user. It takes the values from the ReadInit() method, and creates all the required matrices for the class. This constructor gets called when the user chooses to train the network. Once called, it is this constructor that makes sure every data member of the class gets assigned to a value.

### Parametric Constructor

**Syntax:**

```
        Network::Network(const string &previous_network_filename);
```

**Parameters:**

It takes one parameter, the filename of the previous network that needs to used.

**Description:**

This constructor gets called when the user chooses to train the existing network. It reads in all the values required from the previous_network file and asks user if the user wants to change the hyperparameters. If so, it updates the values of the hyperparameters.

### Overloaded Parametric Constructor

**Syntax:**

```
      Network::Network (const string& previous_network_filename, const string&
      validation_data_filename);
```

**Parameters:**

It takes two parameters, the filename of the previous network that needs to used and the other parameter is the data file name, which needs to be classified.

**Description:**

This constructor assigns the validation_data_filename to the training_data_filename in the class. It also opens the two files passed as the parameters. It takes the required values for classification from the previous network file and creates the required matrices. This constructor gets called when the user chooses to classify the network. Once called, it is this constructor that makes sure every data member of the class gets assigned to a value, if needed and to null, if not needed.

## Method Details

### 1. readInit(string &)

**Syntax:**

```
        void Network :: readInit (const string& previous_network_filename);
```

**Parameters:**

It takes one parameter, which is the name of the file that stores the values of an already trained network.

**Description:**

This method reads in all the required parameters from the file, the name of which is passed in the parameter. This method is responsible to open the file and read in the values of the hyperparameters along with the training_data_filename and the expected_values_filename. It opens the two files and creates the matrices for the data members of the class. Once created, the weights and biases matrices gets assigned with the values from the previous_netwrok file. This method is responsible to check for the validation of the values before assigning them to the data members of the class and display the error message in case of any invalid input.

### 2. readInit()

**Syntax:**

```
        void Network :: readInit ();
```

**Description:**

This method reads in all the required parameters from the console, as entered by the user. Also, it asks the user to enter the name of the test data file and expected values file. This method is responsible to validate the values before assigning them to the data members of the class and display the error message in case of any invalid input. Then, it assigns the values to the data members of the class and returns the address of the name of the test data file.

**Validation Check:**

ReadInit() will be assigning the values to learning_rate, epochs, batch_size, layer_sizes, test_data_file and the expected_values_file. The value of the learning rate user enters has to be less than 1. The layer_sizes has to be valid, so that the layer sizes are always a positive number. The batch_size entered has to be less than the actual data_size. The names of the file entered by the user have to be valid names.

### 3. forwardPropagation()

**Syntax:**

```
        void Network :: forwardPropagation (int mini_batch_index);
```

**Parameters:**

It takes one parameter, which is the mini_batch_index of datatype `int`.

**Description:**

This method extracts the input data from the test_data file and assigns that to the first activations matrix. It then starts a loop that goes upto the last layer of the network and assigns the values to each weighted_inputs and activations matrices.

## 4. backPropagation()

**Syntax:**

```
        void Network :: backPropagation (int mini_batch_index);
```

**Parameters:**

It takes one parameter, which is the mini_batch_index of datatype `int`.

**Description:**

Once the weighted_sums and the activations matrices get assigned, this method checks for the last activations matrix of the network and compares it with the Expected_values matrix of that mini_batch_index. If they match, it increments the num_correct by 1. If they doesn't match, it calculates the errors in the output using the expected values matrix and backpropagates the error. Thus, it assigns the error matrices in the network. Once the error matrices are assigned, it calculates the cost partials and thus, sum_nabla_b and sum_nabla_w gets updated.

## 5. SGD()

**Syntax:**

```
        void Network :: SGD();
```

**Description:**

SGD stands for the idea of Stochastic Gradient Descent to speed up learning process of the network. This method is responsible to complete a forward pass and backward pass on a mini batch and compute the average nabla_b and nabla_w vectors over the batch. It reads in the expected values from the expected_values file and assign the values to the Expected_Values matrix. It starts a loop which goes upto the batch_size. This loop calls the forwardPropagation() and backPropagation() for each mini_batch_index. Once the loops ends, it is responsible to update the values of weights and biases using the sum of cost partials.
At the end of the function, the sum of the cost partials are set to 0 again.

## 6. train()

**Syntax:**

```
        void Network :: train();
```

**Member Description:**

Members named `test_data_size` and `SGD_Calls` are declared in the function of datatype `int`. `test_data_size` gives the size of the training data and `SGD_Calls` gives the number for how many times the SGD() needs to be called to complete an epoch. It is calculated by (size of test data)/(batch size).

**Description:**

This method is responsible for training the network until all training data inputs are exhausted. It performs the training as many times as the number of epochs entered by the user. This method is responsible to assign the values to the test_data_indices initially to the values which represent their respective position in the test_data file. Once initialized, the test_data_indices are shuffled at the start of every epoch. The loop makes sure to jump to the beginning of the test_data file and expected_values file at the start of every epoch. Also, it sets the num_correct to 0 at the start of every epoch. The nested loops are set up, which takes care of the calling SGD() and also, updates the mini_batch_indices vector. At the end of every epoch, the efficiency of the network built is displayed with the number of correct outputs found generated during the learning of the network. As the number of epochs are completed, a file of the network built is made, which stores all the required information of the network that was built.

## 7. update()

**Syntax:**

```
        void Network :: update();
```

**Description:**

Like its name suggests, this method updates the weights and biases matrices of the network. It consists of a loop which goes from the first layer of the netwrok to the last layer, updating the weights and biases matrices in that layer. It requires the cost partials to be calculated before updating the weights and biases matrices.

## 8. classify()

**Syntax:**

```
        void Network :: classify();
```

**Member Description:**

The variables declared inside this function include `ambiguous_data`, `numData` and `biggest`, all of `int` datatype. `ambiguous_data` keeps a track on the number of data which the classifier was not able to classify. The `numData` stores the values for the number of training inputs in the verification file. `biggest` stores the index which has maximum activation value and thus, helps in classification.

**Description:**

Classify() is called when the user wants to classify a specific file. In that case, the value of `numData` is calculated and a loop is started which goes through all the classification data inputs in the file. At the start of the loop, `biggest` is set to 0 and forwardPropagation() is called to assign the values to the weighted_inputs and activations matrices for each classification data input. Once the activations at the ouput layer are assigned, it checks for the biggest value generated. If it finds the biggest value, the training data input is said to be classified.
Otherwise, `ambiguous_data` value is incremented by 1 if no biggest value is found. If biggest was found, it further classifies it into horizontal, vertical or diagonal based on the idex stored in the biggest. Once the loop goes through the complete data set, the ouput is displayed which says the number of data inputs classified according to the different categories and at last displays the accuracy of the classifier based on the classified data out of the complete data set.

## 9. randomizeMatrix()

**Syntax:**

```
        void Network :: randomizeMatrix(matrix<double> input_matrix, double * distribution());
```

**Parameters:**

It takes two parameters, one of them is a matrix and the other is the pointer to the function that performs the distribution.

**Description:**

This method will randomly assign the value to the matrix passed to it. The values generated will be based on the distribution(). It will consist of a nested loop going through each element in the matrix. Each element in the matrix will be assigned with pseudorandom numbers as the distribution() allows.

## 10. createNetworkFile()

**Syntax:**

```
        bool Network :: createNetworkFile();
```

**Return Type:**

This returns a boolean flag, which is true or false.

**Description:**

This method creates a file of the name "Previous_Network_[Date and Time]". If open, the method writes the required values such as the weights and biases matrices, layer_sizes vector and the number of layers to the file created. It returns true once the writing is performed. If the file, with the name passed doesn't open, this method returns false.

## 11. hadamardProduct()

**Syntax:**

```
        matrix& Network :: hadamardProduct (const matrix& M1, const matrix& M2);
```

**Return Type:**

This returns the address of the matrix, which is the result of the Hadamard Product.

**Parameters:**

It takes two parameters of matrix type, which are used as the operands for Hadamard Product.

**Member Details:**

It declares a `matrix M` to store the result of the product.

**Description:**

Hadamard_Product() is a scalar multiplication of two matrices where we obtain a matrix by multiplying an element in one matrix with that respective element in the other matrix. This method checks for the size of the matrices and if they are equal, it performs the Hadamard product on the two matrices and returns the address of the product matrix. If the two matrices passes have different sizes, it displays an error message.

## 12. shuffleDataIndices()

**Syntax:**

```
        void Network :: shuffleDataIndices(vector<T> data_indices);
```

**Parameters:**

It take one parameter which is a vector of the data indices that has to be shuffled.

**Description:**

This method shuffles the data indices passed to it. It consists of a loop which goes through the data indices and performs the operation of shuffling the data.