Matrix:

Since dataset, weights and biases of the neural network is represented as a two dimensional array, we needed an implementation to a two dimensional array with all different operations that can be performed on it.

*Class Matrix ADT:*

enum MatrixType {Identity, Random}

template<typename E>

class matrix

{

private:

int row; int column; E\*\* data;

public:

/\*Constructors and destructor\*/

matrix(int r, int c); //Constructs r x c matrix with nothing in it

matrix(int r, int c, E e); //Constructs r x c matrix filled with e

matrix(int r, int c, MatrixType t);

//Constructs r x c matrix which is either Identity or Random (elements type: integer)

matrix(const matrix& m); //Copy constructor

~matrix(); //Destructor

/\*Visualize data\*/

void print() const; //Prints all elements of matrix

char\* ToString() const; //Converts the matrix into a printable string

/\*Access functions\*/

int Rows() const; //Returns the number of rows in the matrix

int Columns() const; //Returns the number of columns in the matrix

E& access(int r, int c); //Referencing the element [r][c]

matrix operator() (int r1, int c1, int r2 = -1, int c2 = -1) const;

//Getting a sub\_matrix = BIG\_MATRIX(beginning of rows, beginning of columns, end of rows, end of columns);

//Default values of r2 and c2 are the end of rows and columns

/\*Arithmetic operations\*/

matrix operator+ (const matrix m) const;

matrix operator- (const matrix m) const;

matrix operator\* (const matrix m) const; //Element-wise product

matrix operator/ (const matrix m) const; //Element-wise division

matrix operator+ (E n) const;

matrix operator- (E n) const;

matrix operator/ (E n) const;

matrix operator\* (E n) const;

void operator= (const matrix& m); //Assignment operator

matrix dot (const matrix m) const; //Dot product

matrix divide (const matrix divisor) const; //Matrix division

/\*Logic operators\*/

bool operator== (const matrix m) const;

bool operator!= (const matrix m) const;

bool IsIdentity() const;

bool IsIdempotent() const;

//Checks if the dot product between the matrix and itself is the same matrix

bool IsSquare() const;

bool IsSymmetric() const;

bool IsUpperTriangle() const;

bool IsLowerTriangle() const;

/\*Matrix operations\*/

matrix Inverse() const;

matrix CholeskyInverse() const;

matrix SlowInverse() const;

matrix transpose() const;

matrix sum(string choice) const;

//If choice == "row" , result is 1 x column. If choice == "column", result is row x 1.

E sumall() const; //Sums all elements in matrix

E MaxElement() const;

E MinElement() const;

E determinant() const;

void Fill(E e); //Fills all elements with e

matrix LowerTri() const;

matrix LTinverse() const

/\*Special operations\*/

matrix getlog() const; //log(matrix)

matrix square() const; //Element-wise square

matrix Sqrt() const; //Element-wise square root

};

**Dictionary**

To facilitate access to weights and biases matrices of any layer of neural network in any function, we implemented class dictionary which is a list of entries each entry has key (name) and value (matrix).

So we can simply add, remove or get the matrix from dictionary by only knowing its key.

*CharGenerate*

In order to represent the key in dictionary, we needed a function to generate string that concatenate a name (like W or b) to any number (layer number i.e.: 1,2,…).

*Dictionary ADT:*

template <typename K, typename V>

class Entry {

private:

K KEY; //key of entry

V VALUE; //value of entry

public:

Entry(K k = K(), const V& v = V()) : KEY(k), VALUE(v) { } //constructor

const K& key() const { return KEY; } //returns const reference to KEY

const V& value() const { return VALUE; }

//returns const reference to VALUE

void setKey(K k) { KEY = k; } //sets KEY to k

void setValue(V v) { VALUE = v; //sets VALUE to v

};

template <typename K, typename V>

class Dictionary

{

private:

typedef Entry<K, V> ENTRY;

typedef list<ENTRY> Bucket;

typedef typename Bucket::iterator BItor;

private:

Bucket bkt; //list of entries

int SIZE; //size of dictionary

string NAME; //name of dictionary

public:

Dictionary(string name= "YOU DIDN'T GIVE IT A NAME!"); //constructor

int size(); //return the size of dictionary

bool empty();

//return true if the dictionary is empty

bool exist(K key);

//return true if key exists in the dictionary, else it prints err msg

void put(K key, V value);

//put ENTRY(key,value) into dictionary, if the key already exists it prints err msg

void erase(K key); //removes the entry with key, if key does not exist it prints err msg

void replace(K key,V value);

//replaces the value of the entry with the provided key,

if the key does not exist it prints err msg

void clear(); //removes all entries in the dictionary

void print(); //print all entries in the dictionary

//(assuming that the key could be directly printed and the value has its own print function)

void setName(string name); //sets the dictionary name

BItor find(K key);

//returns an iterator to the entry with key, if key does not exist it prints err msg

const V& operator[](K key);

//returns a const refrence the the value of the entry with key

};

**Data Set**

* The data set represents 12 bit even parity problem (the output is high when the number of one`s in input is even) so we implemented a function”InputOutput” to generate the data set.
* InputOutput function: takes as arguments matrix &X,matrix & Y,string ET(Error type)
* X & Y are initialized with zeroes and after this function contain the correct values for the data set.
* X is (12 x 4096) matrix that represents a truth table where each column represents an entry in it.
* Y is (1x 4096) matrix that represents the output corresponding to each column of the input.
* Depending on the error type the values in X&Y are (0 &1) for Cross Entropy or (-1 &1) for square error.
* We implemented two other functions “DataSET” & “SWAP” to generate larger dataset by multiplying the original one by any number n where n>=1 and shuffle the generated dataset to get a random distribution of data set.

**NeuralNetwork:**

*Class ADT:*

class NeuralNetwork

{

private:

layer\* layers;

// Array of layers holding the number of neurons at each layer and its activation

dictionary parameters; // Dictionary containing weights and biases of the network

dictionary cache; // Dictionary containing temporary internal activations of layers

dictionary grades;

//dictionary contains derivates of weights and biases with respect to the cost function

string ErrorType; // Type of cost function or performance index used

string optimizer; // Type of algorithm used

int numOfLayers; // Number of layers

bool momentum; // Indicates whether momentum is used or not

float maxErr;

public:

NeuralNetwork(layer\* mylayers,int L);

// Constructor that initializes the weighs and biases of the network randomly based on the architecture of the network

void test(Matrix X\_test, Matrix Y, string path,bool batchNorm);

// Function that outputs the input training set X, their associated targets Y and the final activations Y\_hat into a text file

void print(); // Prints all parameters of the network

void train(const Matrix& X, Matrix& Y, float alpha, int numOfEpochs, int minibatchSize, string optimizer1,int Numprint, string ET,float lambda,bool batchNorm);

/\*Function that trains the network based on the following arguments:

X: input training set

Y: target associated with the input training set

alpha: learning rate or damping ratio in case of LM optimizer

numOfEpochs: maximum number of iterations

minibatchsize: size of mini-batch (don't care if LM algorithm)

optimizer: the algorithm used to train the network. It's either "GradientDescent", "Adam" or "LM"

Numprint: the number of iterations after which the cost is outputted on the screen

ET: the cost function or performance index. It's either "CrossEntropy" or "SquareErr"\*/

private:

/\*Feed forward\*/

Matrix feedforward(const Matrix& x, layer\* layers,int L,bool batchNorm);

/\*Back propagation\*/

void calGrads(const Matrix& X, const Matrix& Y, const Matrix& Y\_hat, layer\* layers, int L,float lambda,bool batchNorm);

void updateParameters(float& alpha, layer\* layers, int L, int iteration, Matrix& Q, Matrix& g, int m,bool batchNorm);

void BackProp(const Matrix& X, const Matrix& Y, const Matrix& Y\_hat, float& alpha, layer\* layers, int L, int iteration, Matrix& Q, Matrix& g, int m,float lambda,bool batchNorm);

/\* Cost\*/

float CostFunc(const Matrix& y,Matrix& yhat,float lambda,int L);

Matrix costMul(Matrix Y, Matrix Y\_hat);

/\*Classify\*/

Matrix classify(Matrix Y\_hat);

void AccuracyTest(const Matrix& Y,Matrix& Y\_hat\_classified, const Matrix& X,

layer\*layers, int L);

/\*Error\*/

float AbsErr(Matrix\* Y\_hat, Matrix\* Y);

float numOfErrs(Matrix\* Y\_hat, Matrix\* Y);

/\* Store data\*/

void storedata(Matrix X, Matrix Y, Matrix Yhat, string path);

};

Brief explanation of Neural Network functions:

*Constructor:*

* Generating weights and biases W & b matrices for each layer according to layer dimensions defined in array layer where the dimensions of W for any layer L is (number of neurons in L+1) x (number of neurons in L ) and for b is (number of neurons in L+1) x1.
* Initialize W &b with random values then normalize them(make mean=0 and

standard deviation =1).

* To avoid vanishing or exploding weight problems, we used Xavier’s initialization which takes into account the size of the network by multiplying the weights by square root of (x/nl) where x is 1 or 2 depending on the activation used in this layer and nl is number of neurons in this layer.
* Store W & b for each layer in dictionary parameters so we can use them easily in training.

*Train:*

1. If optimizer is Adam, we initialize vdw,sdw,vdb,sdb for each layer and store them in dictionary grades to be used in back propagation.
2. Initializations of parameter cancelling technique (store W & b in prevparameters dictionary to use them instead of the new parameters if the error increased).
3. Loop for number of epochs, in each epoch we divide the training set to number of mini batches and loop for number of mini batches, in each mini batch we implement feedforward to get Y\_hat and backprob to get derivate of weights and update them according to type of optimizer used gradient decent, Adam or LM(Levenberg–Marquardt).

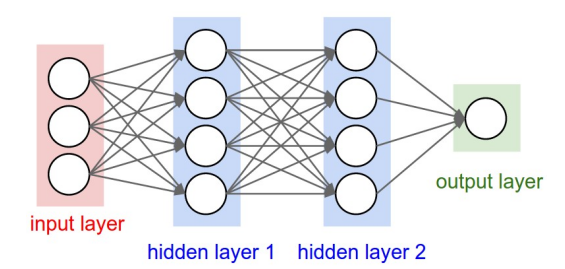
After mini batches loop, we calculate the cost. If parameter cancelling technique is used we compare the cost with the one of previous epoch if it is larger the previous parameter is restored and the learning rate is decreased but if it is smaller store the new weights and increase the learning rate.

1. Classification and accuracy test is implemented by feedforward the entire training set then use Accuracy test function to compare Y\_hat with Y and print the accuracy %

*Test:*

1. Implement feedforwad on the entire training set to get Y\_hat
2. Use store data function to store the resulting Y\_hat along with Y and X foe each pattern.

*FeedForwad:*



1. Loop over the number of layers and compute z=W.Aprev+b ,A=g(z) where g is the activation of this layer.
2. The output is Y\_hat (A of the last layer).

*Activations:*

|  |  |
| --- | --- |
| Name | formula |
| sigmoid | Image result for sigmoid function  Sigmoid: A=1/(1+exp(-z))  Dsimoid: dA= A(1-A) |
| tanh | Image result for tanh function  tanh : A=tanh(z)  dtanh: dA=1-A^2 |
| relu | Image result for relu function  relu: A=0 for z<0 & z for z>=0  drelu: dA=0 for z<0 & 1 for z>=0 |
| leakyRelu | Image result for leaky relu function  leakyRelu: A=0.01\*z for z<0 & z for z>=0  dleakyrelu: dA=0.01 for z<0 & 1 for z>=0 |
| linear | Image result for linear activation function derivative  Linear: A=z  dlinear : dA=z |
| Satlinear-  Satlinear 2&3 are the same but the slope depends on the maximum error | Image result for saturated linear activation function  Satlinear: A=-1 for z<-1 & 1 for z>1 & z for -1<z<1  dsatlinear: dA=0 for z<-1 & 0 for z>1 &1 for -1<z<1 |