European Funds
Knowledge Education Development

European Union
European Social Fund

Computer Programming

Neural Network in C++

Author: Alicja Senczyna

Supervisor:

PhD Piotr Fabian

Introduction
My task was to implement a neural network.
Program consists of 4 files:

- matrices_vectors.h with all mathematical operations
- layer.h with class Layer with implemented forward and backward propagation for single layer, and class OutputLayer which inherits after class Layer with its own backward propagation. It also contains void test_layer() where given functionalities were tested
- neural_network.h with class NeuralNetwork where neural network architecture was built as a list of layers with output layer, void fit() where backward and forward propagation are used on the whole neural network to train it, double** predict() which predicts Y after being given dataset X and double** read_matrix_txt() which use txt file to create matrices to train and predict
- neural_network.cpp where int main() is implemented and all functionalities are used making program works

File **matrices_vectors.h** consists of following functions:

- multiply_matrix_by_vector
- multiply_matrix_by_constant
- multiply_vector_by_constant
- add_matrices
- substract_matrices
- substract_constant_from_vector
- substract_vectors
- add_vectors
- multiply_vectorT_by_vector
- multiply_vectorT_by_constant
- element_wise_multiply
- sigmoid_vector - function sigmoid which returns vector
- derivative_sigmoid_vector
- relu_vector
- derivative_relu_vector
- initialize_layer_weights - initialize layer weights to random numbers from interval <-1, 1>
- initialize_layer_bias - initialize layer weights to random numbers from interval <-1, 1>
- zeros_1d - creates vector with all 0 values
- zeros_2d - creates matrix with all 0 values
- cout_matrix
- cout_vector
- transpose

File **layer.h** consists of: class Layer which have the following variables:

```
int neurons_curr;
int neurons_prev;
double** W; // weights
double* b; // bias
double* z; // neuron value
double* a; // neuron value after activation
double* a_prev; // neuron value after activation previous layer
double* delta; // delta
double* error; // error
double learning_rate;
string activation_function;
```

and two functions: forward and backward propagation. We use activation function to introduce nonlinearity to neural network.

Forward propagation

In forward propagation, we are moving from input to output calculating expected output from a given input using formulas written in code:

```
double* forward(int neurons_prev, double* a_prev) {
            // z -> a_prev * W + b
            // a -> activation(z)
            // cout << "calculate z" << endl;
            this->z = multiply_matrix_by_vector(this->neurons_curr,
neurons_prev, this->W, a_prev);
            // cout_vector(this->neurons_curr, this->z);
            // cout << endl;

            // cout << "calculate a" << endl;
            if (this->activation_function == "sigmoid") {
                  this->a = sigmoid_vector(this->neurons_curr, this->z);
                  // cout_vector(this->neurons_curr, this->a);
            }
            else if (this->activation_function == "relu") {
                  this->a = relu_vector(this->neurons_curr, this->z);
                  // cout_vector(this->neurons_curr, this->a);
            }
            else {
                  throw invalid_argument("Activation function must be
either 'sigmoid' or 'relu'");
            }

            this->a_prev = a_prev;
            // cout << endl << endl;
            return this->a;
```

Backward propagation

In backward propagation, we calculate following values:

- error, which is a difference between calculated value from expected value
- delta, which is element wise multiplication between error and value of derivative of activation function of neuron value
- updated W, which is W minus alpha (learning rate) times transposed neuron values from previous layer (input which goes to current layer) times delta
  W := α * a_prev.T * delta
- updated b, which is b minus alpha (learning rate) times delta
  b := b - α * delta

```
double* backward(int neurons_next, double** W_next, double* b_next,
double* delta_next) {
            // error = W_next * delta_next + b_next * delta_next
            // delta -> error . activation_derivative(a) . -> element
wise multiplications [1, 2, 3] . [2, 2, 2] = [1*2, 2*2, 3*2]
            // W -> W - learning_rate * a_prev.T * delta  -> [1, 2].T *
[1, 2] = matrix
            // b -> b - learning_rate * delta

            //weight
            // cout << "calculate error" << endl;
            double **W_nextT = transpose(neurons_next,
this->neurons_curr, W_next);
            this->error = multiply_matrix_by_vector(this->neurons_curr,
neurons_next, W_nextT, delta_next);
            // cout_vector(this->neurons_curr, this->error);

            //delta
            // cout << "calculate delta" << endl;
            if (this->activation_function == "sigmoid") {
                  this->delta =
element_wise_multiply(this->neurons_curr, this->error,
derivative_sigmoid_vector(this->neurons_curr, this->a));
                  // cout_vector(this->neurons_curr, this->delta);
            }
            else if (this->activation_function == "relu") {
                  this->delta =
element_wise_multiply(this->neurons_curr, this->error,
derivative_relu_vector(this->neurons_curr, this->a));
                  // cout_vector(this->neurons_curr, this->delta);
            }
```

```
            else {
                  throw invalid_argument("Activation function must be
either 'sigmoid' or 'relu'");
            }

            //update weights
            // cout << "update weights" << endl;
            double** w = multiply_vectorT_by_vector(this->neurons_curr,
this->neurons_prev, this->delta, this->a_prev);
            double** w1 =
multiply_matrix_by_constant(this->neurons_curr, this->neurons_prev, w,
this->learning_rate);
            this->W = substract_matrices(this->neurons_curr,
this->neurons_prev, this->W, w1);
            // cout_matrix(this->neurons_curr, this->neurons_prev,
this->W);

            //update bias
            // cout << "update bias" << endl;
            this->b = substract_vectors(neurons_curr, this->b,
element_wise_multiply(this->neurons_curr, this->b, this->delta));
            // cout_vector(this->neurons_curr, this->b);
      }
```

File **neural_network.h** consists of class NeuralNetwork with following elements:

- constructor NeuralNetwork which build neural network architecture using class Layer.

```
Layer* hidden_layers = new Layer[n_layers];
            hidden_layers[0] = Layer(neurons_hidden, input_size, "relu",
learning_rate);
            for (int l = 1; l < n_layers; l++)
            {
                  hidden_layers[l] = Layer(neurons_hidden,
neurons_hidden, "relu", learning_rate);
            }
            OutputLayer output_layer = OutputLayer(output_size,
neurons_hidden, "sigmoid", learning_rate);
```

- void fit() which trains neural network. It does forward propagation on n hidden layers and output layer and then backward propagation on output layer and n hidden layers

```
for (int epoch = 0; epoch < epochs; epoch++)
            {
                  for (int i = 0; i < m; i++)
                  {
                        cout << "\r" << "epoch: " << epoch << " progress:
" << i*20 <<"%";
                        // forward
                        this->hidden_layers[0].forward(this->input_size,
X[i]);
                        for (int l = 1; l < this->n_layers; l++)
                        {

this->hidden_layers[l].forward(this->neurons_hidden, hidden_layers[l -
1].access_a());
                        }
                        this->output_layer.forward(this->neurons_hidden,
hidden_layers[this->n_layers - 1].access_a());

                        // backward

                        this->output_layer.backward(y[i]);
                        this->hidden_layers[this->n_layers -
1].backward(this->output_layer.access_neurons_curr(),
this->output_layer.access_W(), this->output_layer.access_b(),
this->output_layer.access_delta());
                        for (int l = this->n_layers - 2; l > -1; l--)
                        {

this->hidden_layers[l].backward(this->hidden_layers[l+1].access_neurons_
curr(), this->hidden_layers[l + 1].access_W(), this->hidden_layers[l +
1].access_b(), this->hidden_layers[l + 1].access_delta());
                        }
                        cout << endl;
                  }
            }
```

- double** predict() which after given dataset X predicts expected output Y, it does forward propagation.

```
double** predicted = new double* [m];
            for (int i = 0; i < m; i++)
            {
                  // forward
                  this->hidden_layers[0].forward(this->input_size,
X[i]);
                  for (int l = 1; l < this->n_layers; l++)
```

```
                {

this->hidden_layers[l].forward(this->neurons_hidden, hidden_layers[l -
1].access_a());
                }
                this->output_layer.forward(this->neurons_hidden,
hidden_layers[this->n_layers - 1].access_a());

                // print results
                cout << "result for element " << i << ": ";
                predicted[i] = new double[this->output_size];
                for (int n = 0; n < this->output_size; n++)
                {
                      if (output_layer.access_a()[n] >= 0.5)
                            predicted[i][n] = 1;
                      else
                            predicted[i][n] = 0;
                      cout << predicted[i][n] << " ";
                }
                cout << endl;
          }
          return predicted;
```

- double** read_matrix_txt() which reads matrix from txt file and converts into double** matrix to be given as datasets

File **neural network.cpp** which connect all other files to one simple main:

```
int main()
{
      NeuralNetwork nn = NeuralNetwork(1, 1, 1, 2, 0.01);

      double** X_train = read_matrix_txt(6, 2,
"D:/neural_network/neural-network-/raw_matrix_x.txt");
      double** y_train = read_matrix_txt(6, 1,
"D:/neural_network/neural-network-/raw_matrix_y.txt");

      double** X_test = read_matrix_txt(6, 3,
"D:/neural_network/neural-network-/matrix_train.txt");

      nn.fit(1000, X_train, y_train, 6);
      nn.predict(X_test, 6);
}
```

Results

Neural network was given following parameters:

- to train

matrix X:	-1 -2	matrix Y:	0
	-1 -4		0
	4 1		1
	1 2		1
	-3 -1		0
	1 2		1

It was supposed to classify positive (1) and negative (0) values of matrix.

To train such a neural network, only one hidden layer is needed because if we have too many layers, it won't train due to the problem of vanishing gradients when gradients become so small that it keeps weights from changing their values. In the worst possible case, it can stop neural network from further training.

- to predict

dataset X:	4 5 6	predictions: 1
	-2 -4 -1	0
	0 -1 -3	0
	7 9 2	1
	0 -1 -2	0
	8 3 2	1

Results:

```
epoch: 999 progress: 80%
epoch: 999 progress: 100%
result for element 0: 1
result for element 1: 0
result for element 2: 0
result for element 3: 1
result for element 4: 0
result for element 5: 1
```