

CP - Project 2. - SGD Neural Network for simple logical operations

Oskar Forreiter - Makro Sem. 2

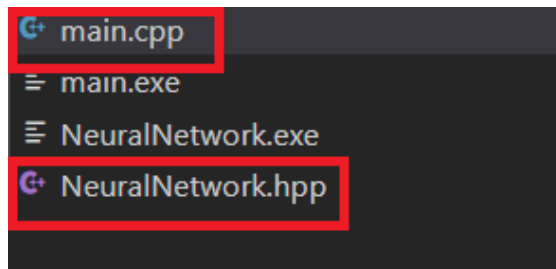
1. Introduction

Following program implements the Neural Network model with SGD that takes the vector of inputs and outputs of a given logical operation (in this example logical “and” operation) and trains the neural network to predict the output.

2. Program structure

Program consists of 2 files:

1. NeuralNetwork.hpp - which implements class of the neural network
2. main.cpp - which instantiates NeuralNetwork object, executes the training procedure and prints the results of model prediction precision



Program structure

3. NeuralNetwork Class implementation

NeuralNetwork class contains several variables required for learning process:

```
class NeuralNetwork {
    float learning_rate;
    vector<vector<float>>> inputs, outputs; // learning set
    // first is an input layer, last layer is an output one
    vector<vector<float>>> nodes, biases;
    vector<vector<vector<float>>>> weights;

    vector<int> topology;
    // topology vector assigned values:
    // [0] - inputs number
    // [1] - layers number
    // [2] - nodes per layer number
    // [3] - outputs number
}
```

Topology vector stands for a dimensions of the neural network

Those variables are initialized in the class constructor as follows:

```
NeuralNetwork::NeuralNetwork(float lr, vector<vector<float>>> inputs,
                             vector<vector<float>>> outputs, vector<int> topology) :
    learning_rate(lr), inputs(inputs), outputs(outputs),
    topology(topology)
{
    initialize_weights_and_biases();
}
```

First we initialize variables through initialization list, then we make call to a function

```
initialize_weights_and_biases();
```

This function initializes biases and weights to a random values according to dimensions specified in the topology vector

```
void NeuralNetwork::initialize_weights_and_biases() {
    for(int i=0; i<topology[1] + 1; i++) {
        vector<float> v;
        vector<vector<float>>> u;
        int z = (i==topology[1] ? 3 : 2); // if last 1
        for(int j=0; j<topology[z]; j++)
        {
            vector<float> s;
            int w = (i==0 ? 0 : 2); // if first layer,
            for(int k=0; k<topology[w]; k++)
            {
                s.push_back(random());
            }
            v.push_back(random());
            u.push_back(s);
        }
        biases.push_back(v);
        weights.push_back(u);
    }
}
```

NeuralNetwork Class implements following functions:

```
void train();

vector<float> predict(vector<float> in);

void initialize_weights_and_biases();

float random() { return ((float) rand()) / ((float) RAND_MAX); }

// activation function
float sigmoid(float x) { return 1 / (1 + exp(-x)); }

// derivative of activation function
float dSigmoid(float x) { return x * (1 - x); }
```

1. **void train()** - trains the neural network on the data set provided in the constructor
2. **vector<float> predict(vector<float> in)** - returns prediction for the given input vector

CP - Project 2. - SGD Neural Network for simple logical operations

Oskar Forreiter - Makro Sem. 2

3. **float random()** - returns random float
4. **float sigmoid(float x)** - returns sigmoid of x
5. **float dsigmoid(float x)** - returns derivative of sigmoid of x

4. Program output and main.cpp execution

Program starts its execution from the **int main()** function in the **main.cpp** file, where the following code is executed:

```
int main() {
    vector<vector<float>> in = {{0,0},{0,1},{1,0},{1,1}};
    vector<vector<float>> out = {{0},{0},{0},{1}};
    float lr = 0.001;
    float c = 0;
    for(int i=0; i<100; i++)
    {
        NeuralNetwork n(lr,in,out,((int)in[0].size(),3,5,(int)out[0].size()));
        n.train();

        int j = interpret_prediction(n.predict({1,1})[0],1);
        if(j==1)
        {
            c+=1.0;
        }
    }
    cout<<"Model accuracy: "<<c/100<<endl;

    return 0;
}
```

It starts with opening the declaring **in**, **out** data vectors and **lr** (learning rate) which will be used to train the neural network. Next it repeats training and prediction process for $i < 100$ times, each time adding 1 to variable **c** for correctly predicted output which is outputted in the end, being first divided by the number of iterations (in this case 100)

Output:

```
[Running] cd "c:\Users\
Model accuracy: 0.99
```

To calculate if prediction was correct we use function:

```
int interpret_prediction(float p,float output) {
    if(output==1)
    {
        return (fabs(p - 1) < fabs(p) ? 1 : 0);
    }
    if(output==0)
    {
        return (fabs(p - 1) > fabs(p) ? 1 : 0);
    }

    return -1;
}
```

which returns 1 if prediction was closer to a real value (either 1 or 0 for logical operations) and -1 if it was exactly in between to cover all cases.

TRAINING ALGORITHM:

```
void NeuralNetwork::train() {
    int epochs = 1000;
    // Iterate through the entire training for a number of epochs
    for (int n=0; n < epochs; n++) {
        // As per SGD, shuffle the order of the training set
        vector<int> trainingSetOrder;
        for(int i=0; i<inputs.size(); i++) (trainingSetOrder.push_back(i));
        vector<int> c = shuffle(trainingSetOrder,inputs.size());
        trainingSetOrder.clear();
        for(int i : c)...
```

We start by declaring the number of epochs and iterating the algorithm **n** times.

Each time we shuffle training set so that SGD is more efficient, and then we begin training loop:

```
// Cycle through each of the training set elements
for (int x=0; x<inputs.size(); x++) {
    int i = trainingSetOrder[x];
    // Compute hidden layers activation
    for(int j=0; j<topology[1]; j++) // layer...

    // Compute output layer activation
    vector<float> d;
    for (int j=0; j<topology[3]; j++) ...
    nodes.push_back(d);
    // Compute change in output weights
    vector<float> deltaOutput;
    for (int j=0; j<topology[3]; j++) { ...
    // Compute change in hidden weights
    vector<vector<float>> deltaHidden;
    for(int j=topology[1]-1; j>=0 ; j--) ...
    // Apply change in output weights
    for (int j=0; j<topology[3]; j++) { ...
    // Apply change in hidden weights
    for(int j=0; j<topology[1]; j++) ...
```

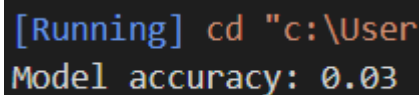
CP - Project 2. - SGD Neural Network for simple logical operations

Oskar Forreiter - Makro Sem. 2

To simplify things we won't discuss each for loop here but only the outer structure of the algorithm which work as follows:

- Compute hidden layer activation
- Compute output layer activation
- Compute change in output weights
- Compute change in hidden weights
- Apply change in output weights
- Apply change in hidden weights

All of which is done according to the structure declared in the topology vector, slowly converging on the solution, which in certain cases can lead to sticking in local minimas and thus rendering model precision as low as 3%

A terminal window with a dark background and light blue text. The first line shows a command prompt followed by a directory change: [Running] cd "c:\User. The second line shows the result of the command: Model accuracy: 0.03.

```
[Running] cd "c:\User
Model accuracy: 0.03
```

Summary

Whereas for most cases the algorithm scores around 90%+ precision, I could not figure out what exactly causes the algorithm to be stuck in local minimas (tried playing with learning rate, epochs etc.). There is a huge probability that multiplication of matrices in the learning algorithm was improperly implemented as working with multidimensional vectors is extremely illegible. Great idea in future implementations would be to use a linear algebra library, such as Eigen, to simplify the operations and make algorithm much slicker, but this is beyond the scope of this project.