EE-559: Deep Learning mini-project 1

Melina CHRYSANTHOU: melina.chrysanthou@epfl.ch
Lorenzo GERMINI: lorenzo.germini@epfl.ch
May 28, 2021

***Abstract*—The objective of this project is to design a mini “deep learning framework” using only Pytorch’s tensor operations and the standard math library. In other words the autograd or neural-network modules shouldn’t be used.**

I. Introduction

This framework imports only the *torch.empty* function and does not use any pre-existing neural-network toolbox provided by python. Furthermore, the autograd module is globally off.

The final framework should provide the necessary tools to:

- build networks combining fully connected layers, Tanh, and ReLU
- run the forward and backward passes
- optimize parameters with SGD for MSE (wasn’t implemented due to time restrains)

II. Methodology and modules

In this section the different components that compose our framework will be presented. Furthermore mathematical principles will be given to better explain each point.

A. Modules

In this section the methods used in the Module of the framework are described.

1) Forward propagation: the input data is fed in the forward direction through the network. Each hidden layer accepts the input data, processes it as per the activation function and passes to the successive layer.

2) Backward propagation: propagates the total loss back into the neural network to know how much of the loss every node is responsible for, and subsequently updates the weights in such a way that minimizes the loss

3) Activation function: decides whether a neuron should be activated or not by calculating weighted sum and further adding bias with it.

B. Losses

Only the MSE loss function was used in this project.
The MSE loss measures the average of the squares of the errors between the output of the network (prediction) and the real value (label).

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2.$$

C. Activation functions

The different activation functions used in this project are explained below.

1) Tanh: the hyperbolic tangent function, which will not change the sign of input but compress it from range $(-\infty, +\infty)$ to range $(-1, 1)$.

$$\texttt{forward(self, input\_)} = \text{tanh(input\_)}$$

$$\texttt{backward(self, x)} = 1 - \text{tanh}^2\text{(x)}$$

2) Sigmoid: a bounded, differetiable function that is defined for all real input values and has a non-negative derivative at each point.

$$\texttt{forward(self, input\_)} = \frac{1}{1 + e^{-\text{input\_}}}$$

$$\texttt{backward(self, x)} = \text{sigmoid}(x)(1 - \text{sigmoid}(x))$$

3) ReLU: the Rectified Linear Unit Activation Function (ReLU) is the most used activation function, which has no shortcomings with gradient vanishing.

$$\texttt{forward(self, input\_)} = \max(0, \text{input\_})$$

$$\texttt{backward(self, x)} = \begin{cases} 1, & \text{if x} > 0 \\ 0, & \text{otherwise} \end{cases}$$

III. Implementation

A. Data generation

The implemented framework generates a training and test set of 1,000 points sampled uniformly in $[0, 1] * [0, 1]$. Each point is labeled with 0 if its positioned outside the disk centered at (0.5, 0.5) of radius $\sqrt{1/2\pi}$, and 1 if its positioned inside.

B. Network structure

The network is composed of five layers:

- two input units
- one output unit
- three hidden layers of 25 units

Firstly, the model is trained using the Mean Square Error (MSE) and then the trained model is tested with the three previously mentioned activation layers.
The model is trained according to the following settings:

- Rounds = 7
- Learning rate of the neurons = 0.01
- Momentum factor of the neurons = 0.15

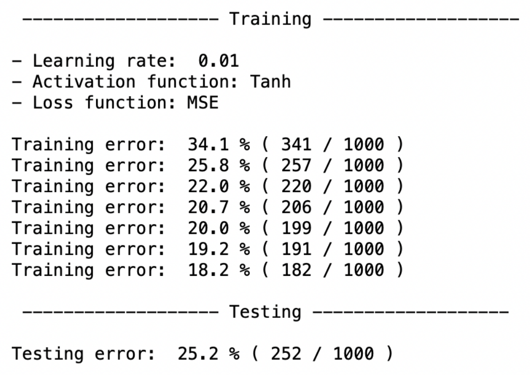
IV. Results

In the following section the results for both the training and testing sets depending on the activation functions used will be presented and commented briefly.

As a general observation we see that as the network is getting trained, the error seems to decrease.

A. Activation Function: Tanh

Here we observe the results obtained wit the Tanh activation layer. As one can notice, the error decreases as the model is trained and we see that the training gets quite low on the final round. However the testing error is much higher than the last training error.

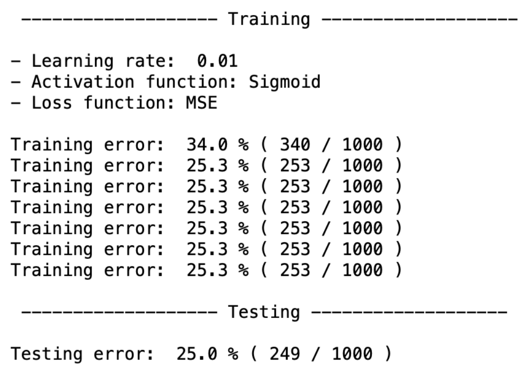


```
 ------------------- Training -------------------

- Learning rate:  0.01
- Activation function: Tanh
- Loss function: MSE

Training error:  34.1 % ( 341 / 1000 )
Training error:  25.8 % ( 257 / 1000 )
Training error:  22.0 % ( 220 / 1000 )
Training error:  20.7 % ( 206 / 1000 )
Training error:  20.0 % ( 199 / 1000 )
Training error:  19.2 % ( 191 / 1000 )
Training error:  18.2 % ( 182 / 1000 )

 ------------------- Testing -------------------

Testing error:  25.2 % ( 252 / 1000 )
```

Fig. 1. Results of training/testing with a Tanh activation function

B. Activation Function: Sigmoid

For the Sigmoid activation function we see that the training error at the beginning is as high as the error of the Tanh activation layer. However it does not decrease as much. We see that it goes down and the stabilises to 25.3%. Finally the testing error is slightly lower than the training error.

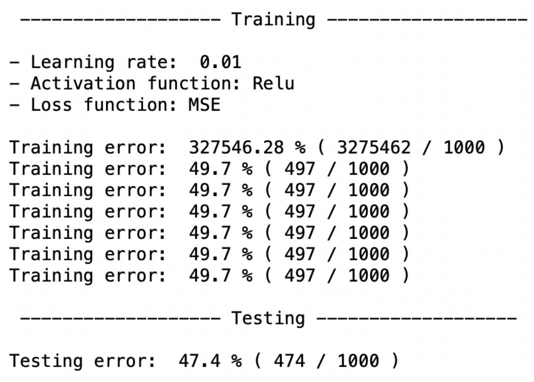


```
 ------------------- Training -------------------

- Learning rate:  0.01
- Activation function: Sigmoid
- Loss function: MSE

Training error:  34.0 % ( 340 / 1000 )
Training error:  25.3 % ( 253 / 1000 )
Training error:  25.3 % ( 253 / 1000 )
Training error:  25.3 % ( 253 / 1000 )
Training error:  25.3 % ( 253 / 1000 )
Training error:  25.3 % ( 253 / 1000 )
Training error:  25.3 % ( 253 / 1000 )

 ------------------- Testing -------------------

Testing error:  25.0 % ( 249 / 1000 )
```

Fig. 2. Results of training/testing with a Sigmoid activation function

C. Activation Function: ReLU

Finally, when using the ReLU activation function we see that we have a huge training error at the beginning. The error then stabilises, as for the Sigmoid activation function, at 49.7%. Furthermore, even if the testing error is lower than the training error we note that it is significantly high.
We know that in the ReLU activation function only a certain number of neurons are activated and therefore it is far more computationally efficient compared to the sigmoid and Tanh function. This result could thus let us think that even though it is computationally more efficient it has a higher error. However no optimizer has been used here in our project (time didn't allow it). The optimizer could have changed significantly the training and test error. Another reason that the errors are so high could be due to under fitting. The classifier learned on the training set is not expressive enough to even account for the data provided.



```
 ------------------- Training -------------------

- Learning rate:  0.01
- Activation function: Relu
- Loss function: MSE

Training error:  327546.28 % ( 3275462 / 1000 )
Training error:  49.7 % ( 497 / 1000 )
Training error:  49.7 % ( 497 / 1000 )
Training error:  49.7 % ( 497 / 1000 )
Training error:  49.7 % ( 497 / 1000 )
Training error:  49.7 % ( 497 / 1000 )
Training error:  49.7 % ( 497 / 1000 )

 ------------------- Testing -------------------

Testing error:  47.4 % ( 474 / 1000 )
```

Fig. 3. Results of training/testing with a Relu activation function

V. Conclusion

The activation function has a great impact on the performance of the network. They are a critical part of the design of a neural network. The choice of activation function in the hidden layer will control how well the network model learns the training dataset. However, it is not the only parameter to take in to account when building a network. A good optimizer can make a difference in the output of the network as they change the attributes of the neural network such as weights and learning rate to reduce the losses. Furthermore, building a network from scratch demands great work and thank god for Pytorch and TensorFlow we don't have to do it every day.