# EE559 Deep Learning
# Mini-project #2 Report

Mehdi Akeddar (261344), Hugo Birch (261684), Louis Piotet (260496)
*EPFL - Switzerland. The 22nd of may, 2020*

***Abstract*—This report is a description of our approach for the second mini-project in the Deep Learning course given at EPFL, Lausanne. The project aims to create a mini deep learning framework and use it for a simple classification task with randomly generated data.**

**PROJECT #2**
**MINI DEEP-LEARNING FRAMEWORK**

This miniproject focuses on the implementation of a simple deep-learning framework from scratch using the bare minimum of what PyTorch as to offer: just basic tensor operations. The use of its autograd abilities is completely forbidden.

## I. FRAMEWORK IMPLEMENTATION

Each module that compose our framework derives from the suggested class model in the project guidelines with the basic `forward()`, `backward()` and `param()`. Each module is implemented as a child class that redefines these functions. They also have their own constructor that allows the instances of these modules to know key informations that will be used for back-propagation. Each module can also describe itself briefly to the user with a call to their `info()` method.

### A. Linear Module

This module implements a fully connected linear layer. Like PyTorch, the user has the possibility to set the biais to zero and only deal with weights. The initialization of the weights and bias of this layer is done by a normal distribution with zero mean and a standard deviation computed using the input and output dimension (Glorot, Bengio (1)).

$$\sigma = \frac{2}{\sqrt{(D_{in} + D_{out})}} \tag{1}$$

Where $D_x$ is the input or output dimension.

### B. Sequential module

The sequential module serves as a container for multiple modules chained together. The framework supports nested Sequential modules. This is implemented in a kind of a recursive manner, where the sequential module features both a forward and backward function. Both consist of a loop that calls the forward or backward of each elements that compose it in the proper order. This is a solid base implementation that enable the possibility to reuse nets in parts of larger networks and possibly different ones too. It can also be useful for network that will use weight sharing, like on mini-project #1 where the same network is used more than once, in a parallel manner.

### C. Activation Modules

Three basic activation functions that inherit from *Module* are built-in, these are `ReLU()`, `sigmoid()` and `tanh()`.

### D. MSELoss class

This implements the Mean Square Error loss. It is defined as:

$$J(x, l) = \frac{1}{n} \sum (x_i - l_i)^2 \tag{2}$$

where n is number of element contained in x (mini-batch size in this case). For simplicity, the loss inherits from *module* as well, where forward computes the actual loss and backward computes the derivative of the loss.

### E. Optimizer

Every optimizer inherits from a generic Optimizer class. The function that will inherit any child of this class are rather basic such as `zero_grad()` which sets the derivative of each parameter of the net to zero. The `step()` method performs a gradient descent step, this one is redefined for each optimizer. A SGD optimizer has been implemented using the recommendation from the PyTorch website. It can be either with or without a momentum (pass it as zero or None to skip the additional computation). The SGD optimizer does not get to know the model, it is just fed with the model's parameters (and their associated gradient accumulator) upon being instantiated.

## II. DATA DESCRIPTION

The data provided consists of 1000 points sampled uniformly between 0 and 1. The points outside the disk of radius $\frac{1}{\sqrt{2\pi}}$ and center (0.5,0.5) have a label 0 and the points outside the disk have a label 1. The dataset is normalized before being fed to the network. The network is trained with a balanced dataset and then tested with a set that may or may not be balanced.

## III. NETWORK DESCRIPTION

The network used for testing our implementation consists of 1 input layer, 3 hidden layers of 25 units in width, and one output layer. The activation function used was the ReLU after every layer.

### A. Implementation

To illustrate the capabilities of our implementation we defined the overall network using nested sequential networks. To be more precise, we defined two different sequential model comprising:

- The first part - Linear layers $2 \times 25$ and $25 \times 25$ (no bias) with rectified linear units at the output of both.
- The second part - Linear layers $25 \times 25$ (no bias) and $25 \times 2$ with ReLU activation functions as well at the output of both.

The two parts are then combined together with another sequential. This will serves as the model to train. Two layers had their bias deactivated to showcase the possibilities of the framework.

### B. Network self description

Using the `info()` function of a model, the program can remind the user the structure they are working with:

```
In [8]: model.info()
|x Sequential:
 |x Sequential:
  | Linear Layer, in: 2, out: 25, biais: True
  | Rectified Linear Unit
  | Linear Layer, in: 25, out: 25, biais: False
  | Rectified Linear Unit
  V
 |x Sequential:
  | Linear Layer, in: 25, out: 25, biais: False
  | Rectified Linear Unit
  | Linear Layer, in: 25, out: 2, biais: True
  | Rectified Linear Unit
  V
 V
```

Figure 1: Implemented Sequential solution, describing itself upon calling its `info()| method`.

This is especially helpful to have have a quick overview of a potentially very complex network and for debugging. Lastly, it's very common for people to take over the work of someone else. This allow them to get a glimpse of what was accomplished without digging in the code too deeply.

## IV. Tests

### A. Performance

On the best runs, the network manages to achieve less than 7% error for both the testing and training sets. 100 epochs may seem a bit much, we can get similar error rates with half less on a common basis. Plotting the test and train error over each epoch gives the lines observed on fig. 2.

As the dataset lies in a 2D space, it's easy to plot it and see which sample classifies incorrectly for both the training and testing set. This yields figures 3 and 4.

Finally, the time to train the model with 50 epochs is quite low. On a 5 years old 3.1GHz i7 CPU, we achieve a total training time that is generally below 600 milliseconds. As printing anything on the terminal consumes a lot of time, removing any prints results in another 50 milliseconds gain.

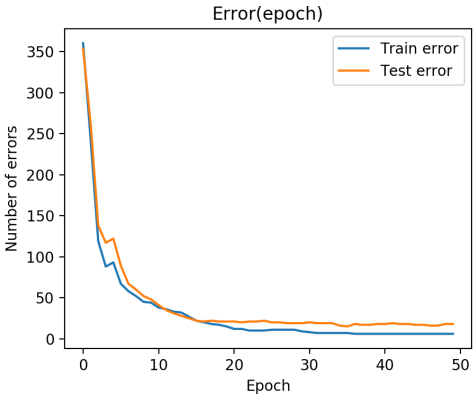


Figure 2: Train and Test Error with 50 epochs.

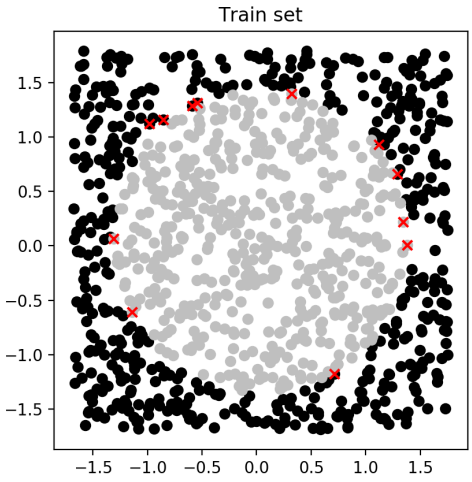


Figure 3: A typical result on the training set. Normalized data.

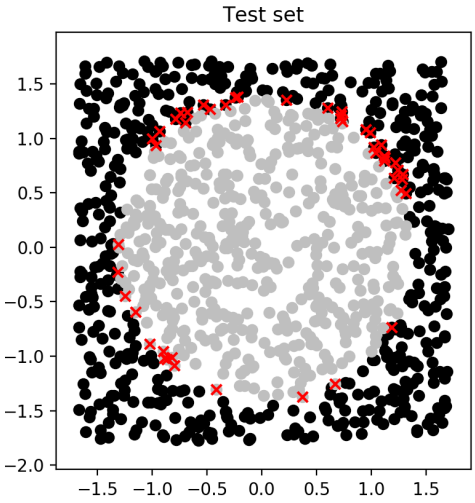


Figure 4: A typical result on the testing set. As one can see, the points that are incorrectly classified lie on the border. Normalized data.

### B. Known problems and limits

On some trials, the gradient might vanish and results in the system learning poorly or not at all. There are currently no implemented solution to work around this. This problem appears to be strongly linked with the initial parameters values set using a normal law. Depending of the distribution, the model may underperform or not perform at all. There are no workaround, the user is required to perform multiple folds and pick the model that performs best according to their needs.

### C. Testing the scripts

The provided script is run without argument. Plots and execution time can be enabled by changing two booleans at the beginning of the script. One will enable plots similar to figures 2, 3, 4. The other will compute and display the training time of the network.

## V. Future development

The framework developed for the project ticks every requested functionalities. It can also easily be extended with new modules, whatever it'd be a type of layer, an activation function, an optimizer or loss. There's one thing that is currently not supported, reusing the exact same instantiation of a net in the same pass. This is due to the layer keeping track solely of the last applied input. However, this can be easily fixed by adding a push and pop behaviour that stores the intermediate input of each layer. This can be implemented with a Python list, for example. This change would also require a method to switch the model to be in either training or testing, to avoid pushing endlessly values that comes from the testing process.

## VI. Conclusion

The second mini-project illustrated fairly well that implementing a standard deepnet framework with some basic functionalities was not relying on black magic but could be accomplished by most. Yet, making a *good* framework that could offer some more advanced possibilities but also could be extended without restarting from zero required some particular considerations and thinking further ahead. In a way, it was inspiring but reinventing the wheel sometime is not particularly exciting. But conversely, this is an essential step to pass on the torch.

## Bibliography

[1] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," 2010.