# Deep Learning - Miniproject 2

Olesia Altunina (285467), Mauro Pfister (235440), Joey Zenhaeusern (226652)

***Abstract*—This report is a description of our approach to implement a simple deep-learning framework from scratch and particularly without using PyTorch's automatic differentiation. The framework contains all the necessary tools to create, train and evaluate a simple network. The functionality is demonstrated by building a classification network for a randomly generated dataset.**

## I. Introduction

## II. Framework implementation

In order to implement our own mini deep-learning framework MyTorch we chose a structure based on the suggestion provided in the project description. All modules are implemented as classes which inherit from a parent class `Module`. They need to override the `forward()`, `backward()` and `param()` methods.

MyTorch supports batch processing which significantly decreases training times as well as prediction times on large networks. Currently, we do not support multiple forward passes since the inputs are not saved into a buffer. While it is still possible to run several forward passes one should keep in mind that the backward pass will only calculate the gradients with respect to the last applied input.

### A. Linear module

The linear module implements a simple fully connected layer with or without bias. Following the suggestion of Glorot and Bengio [1], weights and biases are initialized with a normal distribution of zero mean and standard deviation $\sigma$ given by

$$\sigma = \sqrt{2/(n_{in} + n_{out})}, \tag{1}$$

where $n_{in}, n_{out}$ denote the input and output features respectively.

If the back propagation is carried out on a batch of samples, the gradients of the weights and the biases are averaged over the whole batch. This is in accordance with the official PyTorch implementation and ensures that the gradient norms do not grow with the batch size.

### B. Sequential module

The sequential module takes a chronological list of other modules as input and groups them in one module for convenience. The forward pass sequentially executes the forward passes of the individual modules. The backward pass does the same but in reversed order.

### C. ReLU and Tanh module

Both ReLU and Tanh modules have no parameters.

### D. MSE loss module

The `LossMSE` class implements a mean squared error loss function defined as follows:

$$l(x, t) = \frac{1}{n} \sum_{1}^{n} (x_i - t_i)^2, \tag{2}$$

where $n$ defines the dimension of $x$ and $t$. Note that we currently do not support the computation of the loss over a batch.

### E. Optimizer

Additionally to the required modules above we also defined an optimizer class `Optimizer` that can be used to implement different optimizers. It takes the parameters of a model as input and contains a `step()` method to perform a gradient step on the model parameters. A second method `zero_grad()` can be used to set the gradients of all optimized parameters to zero.

Currently, only a stochastic gradient descent optimizer `SGD` is available,that can be used with or without momentum. The implementation of momentum is based on the one in PyTorch which differs from Sutskever et. al. [2]. The update of parameter $p$ is defined as follows

$$\begin{aligned} v &= \rho v + g \\ p &= p - \alpha v, \end{aligned} \tag{3}$$

where $\alpha$, $\rho$ and $g$ denote learning rate, momentum and parameter gradient respectively.

## III. Framework validation

To ensure the correctness of our implementations we wrote a series of unit tests which compare the functionalities of our code to the corresponding PyTorch functionalities. Currently the forward and backward passes of `Linear`, `ReLU` and `Tanh` modules can be tested.

Additionally, we created the same network architecture once MyTorch and once in PyTorch and initialized all model parameters to a value of $10^{-6}$. This ensures that both models have exact the same initial condition. Using this initialization the total training loss per epochs does not differ by more than $10^{-6}$ and the training and test accuracies match perfectly. Therefore we conclude that within the limitations of our framework, the functionality is equivalent to PyTorch.

## IV. Data description

According to the task description the training and test data sets consist each of 1000 points sampled uniformly in $[0, 1]^2$. Class 0 is defined by all the points outside the disk with radius $1/\sqrt{2\pi}$ and center $(1/2, 1/2)$ while class 1 denotes all the points inside that disk (see Figure 1).

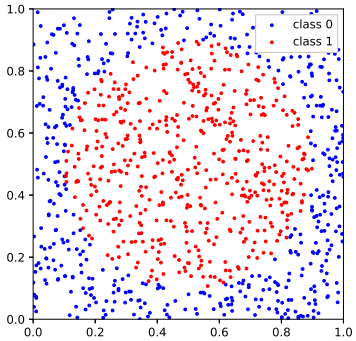


Figure 1: Visualization of data set.

## V. Network description

The network used for classification of the data set consists of two input units, three hidden layers of 25 units and two output units. We chose to use ReLU as activation functions for all layers.

## VI. Performance evaluation

When training the model we noted that the error and loss values would vary by quite a lot. Figure 2 and Figure 3 depict the test and train error comparison of the same model trained in MyTorch and Pytorch over 25 epochs. We suspect that this variation is due to the random nature of the data (which is newly generated on every run) and the randomness in the weight initialization. Overall our framework performs very similar to the python reference implementation.

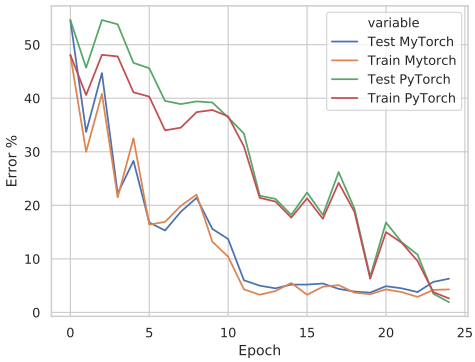


Figure 2: Test and train error over 25 epochs.

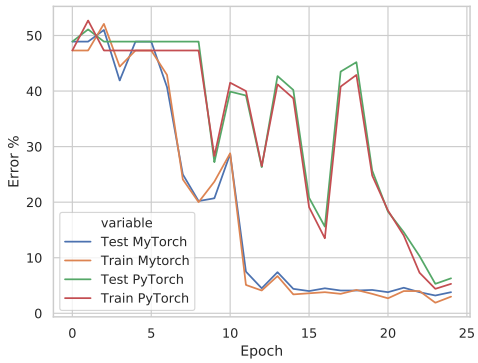


Figure 3: MyTorch vs. PyTorch loss over 25 epochs.

| | Test error (mean) | Test error (std) |
|---|---|---|
| MyTorch | 4.0 | 6.5 % |
| PyTorch | 5.3 | 7.9 % |

Table I: Test error results after running model 10 times

In Table 1 we can see the test error results after running the model 10 times. Surprisingly our model manages to achieve a slightly lower test error in the mean as well as slightly lower standard deviation.

## References

[1] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, Y. W. Teh and M. Titterington, Eds., vol. 9. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. [Online]. Available: http://proceedings.mlr.press/v9/glorot10a.html

[2] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International conference on machine learning*, 2013, pp. 1139–1147.