Deep Learning - Project 2

Manuel Leone, Giorgio Savini Zangrandi
School of Computer and Communication Sciences - EPFL

***Abstract*—In this project we created a small Deep Learning (DL) framework, using Pytorch's tensor operations but no `autograd` machinery, that enables easy differentiation. The implementation contains all the basic modules to build, train and test a simple DL network. Our framework's performances are later evaluated by implementing a basic neural network with 3 hidden layers and using it for a geometric classification task, with a randomly generated dataset. Finally, we compared our implementation to an equivalent one in Pytorch and found similar performance under the same conditions.**

I. Introduction

The purpose of this project was to implement a small DL framework, with a structure similar to Pytorch's one (we called it `myNN`). To achieve this, we were allowed to use all tensor operations provided by Pytorch itself, but **not** the `autograd` function, or any of the already-implemented modules. Moreover, we needed to implement an optimizer to train our network. The request was to implement a **Stochastic Gradient Descent** (SGD), which we slightly improved by adding the Momentum method. All the details about the different model implementations can be found in Section II.

To test our framework, we generated a network and trained it to classify 1000 2D points. Different rounds of validation were done on these data, by generating a new random sample at each validation round instead of performing a k-fold cross validation. The details are in Section III.

The final results, comparisons and discussion are presented in Section IV and Section V.

II. Implementations

The mini-DL framework implementation followed roughly the suggested implementation of a `Module` superclass, with some insight gained by analyzing the literature and Pytorch implementations. Among the main features of `myNN` with respect to the minimum requirements there are: the possibility to use **batch processing** for training, which dramatically increases the performances, the use of **Momentum** for SGD and a wise **parameter initialization**.

The implementation is based on the well-known **back-propagation** algorithm, which allows to compute the model output (*forward pass*) and the gradient with respect to the loss of each parameter (*backward pass*) to use it in the SGD.

To fulfill the requirements, each module extended the `Module` superclass, implementing the a `forward` and a `backward` function at least. Some modules may also use a `param` function to returns the parameters and respective their gradients (to optimize them) and a `zero_grad` function to set all the gradients to zero.

One thing to note is that we accumulate gradients in the `backward` function (if needed by the module) because this allows to perform full gradient descent if the user so wishes. For stochastic gradient descent, the parameters' gradients can be reset to zero at every iteration (through each sample or batch) by using the `zero_grad` function.

A. Linear Module

The `Linear` module implements a fully-connected layer. It has two vector parameters, **weight** $w^{(l)} \in \mathbb{R}^{n_{out}^{(l)} \times n_{in}^{(l)}}$ and **biases** $b^{(l)} \in \mathbb{R}^{n_{out}^{(l)}}$, where $n_{in}^{(l)}$ and $n_{out}^{(l)}$ represent the layers' input and output units.

A wise parameter initialization is fundamental, we tried to use both the *Xavier initialization* [1] and the one used in Pytorch. We found similar performances for both of them, so we decided to use the Pytorch one to enable better comparison with the `torch.nn` package. The initialization is shown in the following equation 1:

$$\sigma = \frac{1}{\sqrt{n_{in}^{(l)}}},\ w^{(l)} = \mathcal{U}(-\sigma, \sigma),\ b^{(l)} = \mathcal{U}(-\sigma, \sigma) \tag{1}$$

Where $\mathcal{U}(a, b)$ is a uniform distribution in a, b.

The `forward` function returns $s^{(l)} = \left(w^{(l)}\right)^T x^{(l-1)} + b^{(l)}$. In the `backward` function, besides returning $\frac{\partial \ell}{\partial x^{(l-1)}}$ (as shows in equation 2) it also accumulates the gradients of the loss with respect to weights, $\frac{\partial \ell}{\partial w^{(l)}}$, and the biases, $\frac{\partial \ell}{\partial b^{(l)}}$. Equations 3 and 4 from the back-propagation algorithm show how these quantities are derived.

$$\frac{\partial \ell}{\partial x^{(l-1)}} = \left(w^{(l)}\right)^T \frac{\partial \ell}{\partial s^{(l)}} \tag{2}$$

$$\frac{\partial \ell}{\partial w^{(l)}} = \frac{\partial \ell}{\partial s^{(l)}} \left(x^{(l-1)}\right)^T \tag{3}$$

$$\frac{\partial \ell}{\partial b^{(l)}} = \frac{\partial \ell}{\partial s^{(l)}} \tag{4}$$

Finally, the `param` function of this module returns a list of tuples of the form `[(weights, dweights), (bias, dbias)]`, as defined by the algorithm.

B. Activation Functions

The activation functions we implemented are `Tanh` and `ReLU`. These are quite straightforward modules, that apply their respective functions point-wise to their inputs in the `forward` function. Given that they have no parameters, the `param` function returns an empty list and the `backward` function does not perform any gradient accumulation.

For the `backward` function, we must consider that the input to the module is $s^{(l)}$ (the linear layer's output) and the output is $x^{(l)}$. Given $\frac{\partial \ell}{\partial x^{(l)}}$, we can thus obtain $\frac{\partial \ell}{\partial s^{(l)}}$ using Equation 5, where $\odot$ is the point-wise Hadamard product and σ is the non-linear function implemented by the module (`tanh` or `relu`).

$$\frac{\partial \ell}{\partial s^{(l)}} = \frac{\partial \ell}{\partial x^{(l)}} \odot \sigma'\left(s^{(l)}\right) \tag{5}$$

Finally, the derivative $\sigma'(x)$ was obtained as shown in Equations 6 and 7.

$$\tanh'(x) = 1 - \tanh^2(x) \tag{6}$$

$$\text{relu}'(x) = \begin{cases} 0, & x \leq 0 \\ 1, & x > 0 \end{cases} \tag{7}$$

C. MSE Loss

The `LossMSE` module implements the **MSE Loss**, given a prediction p and a target t with identical shapes. In order to emulate the behavior of Pytorch for this module and correctly enable batch computation, we first calculated the total number of elements N in each tensor (Equation 8), then divided both the loss and the derivative of the loss with respect to the input by this quantity (Equations 9 and 10).

$$N = \prod_i s_i, \; s_i \in \text{shape}(p) \tag{8}$$

$$\ell(p,t) = \frac{1}{N} \|p - t\|_2^2 \tag{9}$$

$$\frac{\partial \ell}{\partial p} = \frac{2}{N}(p - t) \tag{10}$$

Given that the module has no parameters, the `param` and `backward` functions behave like in the activation modules described in Section II-B.

D. Sequential

The `Sequential` module generates a network by **concatenating input modules** sequentially, much like `torch.nn.Sequential` in Pytorch. The `forward` and `backward` functions simply traverse the submodules (in forward and reverse order respectively) and call their corresponding functions.

The `param` function, similarly, traverses all submodules, collects the return values of their `param` functions and returns everything as a list, where each element corresponds to the output of `param` for a given submodule.

E. SGD

In order to train our network, we implemented an optimizer capable of performing stochastic gradient descent (the `SGD` class). `SGD` takes as inputs the parameters of the module to optimize and modifies them in-place, much like Pytorch's `torch.optim.SGD`. The user must specify the **learning rate** η and may also provide a **momentum factor** μ. An optimization step is performed by calling the `step` function, which will use the update rule in Equation 11 when $\mu = 0$ (default) and the one in Equation 12 when $\mu \neq 0$ (where r is the parameter to optimize). Equation 12 is derived from Pytorch's implementation of the momentum factor. Note that this implementation is slightly different from the original one proposed by Sutskever et. al. [2].

$$r^{(t+1)} = r^{(t)} - \eta \frac{\partial \ell}{\partial r} \tag{11}$$

$$\begin{cases} v^{(t+1)} = \mu v^{(t)} + \frac{\partial \ell}{\partial r} \\ r^{(t+1)} = r^{(t)} - \eta v^{(t)} \end{cases} \tag{12}$$

F. Training

```
model = # instantiate model                                    1
criterion = LossMSE()                                          2
optimizer = SGD(model.param(), eta, momentum)                  3
for e in epochs:                                               4
    for data, target in batches:                               5
                                                               6
        prediction = model(data)                               7
        loss = criterion(prediction, target)                   8
                                                               9
        # optimization step                                    10
        model.zero_grad()                                      11
        model.backward(criterion.backward())                   12
        optimizer.step()                                       13
```

A simplified version of our algorithm is shown above, where we would like to point out 3 lines in particular:

- **Line 11** ensures that the algorithm performs stochastic (rather than full) gradient descent, by setting all gradients to zero for every batch instead of accumulating them.
- **Line 12** performs the back-propagation step and updates all the gradients in the model. The input to the model's `backward` function is the gradient of the loss with respect to the model's output, $\frac{\partial \ell}{\partial p}$ (see Equation 10).
- **Line 13** performs the gradient descent step, where the optimizer updates all the parameters in the model.

III. Data and Validation

To test our framework, we generated a training and a test set of **1000 2D points** in $[0,1]^2$ and classified them with a 1 if they were within a circle of radius $\frac{1}{\sqrt{2\pi}}$ centered at $(0.5, 0.5)$, and 0 otherwise. This allowed for an equal split between the 2 different labels, as the circle has area $1/2$ and is entirely contained within $[0,1]^2$.

For training, we used a neural network with the following characteristics:

- 2 input units, for the (x, y) coordinates of each point.
- 2 output units, for one-hot labeling (1 put in the class' index to predict).
- 3 hidden layers of 25 units each.

We used the same activation function for every hidden layer, i.e. we did not mix `Tanh` and `ReLU` modules but used either one or the other throughout the model.

Our hyper-parameter space for this model was quite small, as it only contained the learning rate η and the momentum

factor μ. Nevertheless, we performed validation over these two quantities to find the best combinations, which are shown in Table I. The validation was performed by generating a new random dataset for each run and averaging the model results, in order to avoid overfitting a single dataset.

Activation	η	μ
`ReLU`	0.1	0.6
`Tanh`	0.1	0.9

TABLE I
BEST VALUES OF η AND μ FOR DIFFERENT ACTIVATION FUNCTIONS.

Figure 1 shows an example of how our model classified the test dataset using the `Tanh` activation, after 75 epochs.

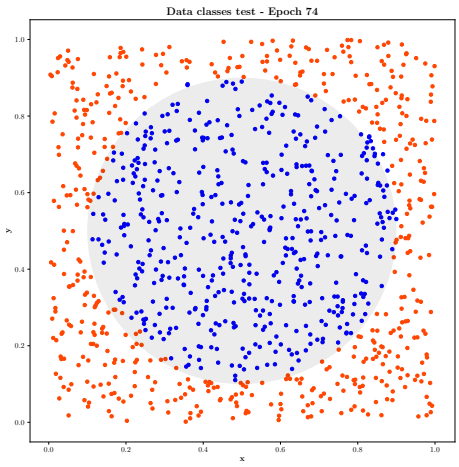

Fig. 1. Classification with `Tanh` activation after 75 epochs. The shaded area is the ground truth. The blue points are classified as 1, the orange ones as 0.

IV. RESULTS

We were able to train our model for 75 epochs without observing any overfitting. This is to be expected, as the input data did not have any noise, so the test error should theoretically only decrease with the training error.

To have a baseline for comparison, we compared our model with an equivalent one implemented in Pytorch and found that the performance was very similar under the same conditions. In order to have a fair comparison and reproducible results, we fixed the seed in the two models and generated the same set of `n_runs` random datasets of 1000 points for both of them. Table II summarizes the results for our training and test datasets of 1000 points each, averaged over 15 runs. The same hyper-parameters and epochs were used for both implementations.

The results show that our model performs the same as Pytorch, being able also to slightly surpass its performance when used with the `Tanh` activation. Figures 2 and 3 show the training and test loss averaged over 15 runs, for both models tested. We can see that our model behaves very similarly to Pytorch across the 75 epochs of training, confirming again the implementation's correctness. Similar plots can be found in the Appendix A, showing the error decreasing over the epochs, and how the model performs using `ReLU`.

	Activation	Train/Test	Mean #Err.	Std. Dev.
myNN	`ReLU`	Train	42	27
	`ReLU`	Test	47	24
	`Tanh`	Train	34	9
	`Tanh`	Test	37	11
Pytorch	`ReLU`	Train	43	26
	`ReLU`	Test	48	24
	`Tanh`	Train	34	11
	`Tanh`	Test	38	12

TABLE II
MEAN NUMBER OF ERRORS AND STANDARD DEVIATION OVER 15 RUNS.

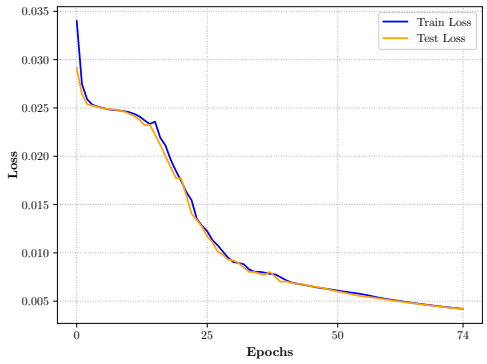

Fig. 2. Training and test loss for our model, averaged over 15 runs, with `Tanh` activation.

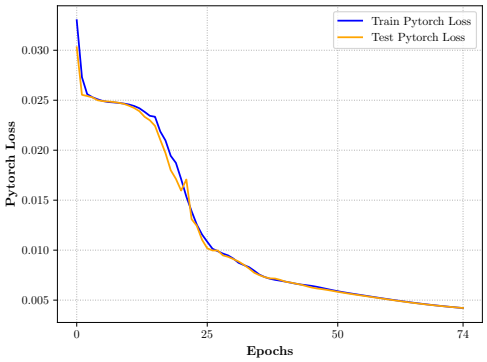

Fig. 3. Training and test loss for the Pytorch model, averaged over 15 runs, with `Tanh` activation.

V. CONCLUSION

In this project we implemented a small DL framework inspired by Pytorch, using tensor operations, but without any of the `autograd` functionality. We implemented fully-connected layers, `ReLU` and `Tanh` activations, MSE loss, an SGD optimizer and a `Sequential` module, using the back-propagation algorithm.

Finally, we tested our framework with a small network on a geometric classification task and compared it to an equivalent implementation in Pytorch. Overall, we found that the two models performed very similarly, meaning that, within our limited scope, our framework achieved the objective of implementing a mini DL framework from scratch. Moreover, our implementation's high modularity makes really easy to add new modules or introduce a new optimizer in the future.

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 *Proceedings of the 30th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, S. Dasgupta and D. McAllester, Eds., vol. 28, no. 3. Atlanta, Georgia, USA: PMLR, 17–19 Jun 2013, pp. 1139–1147. [Online]. Available: http://proceedings.mlr.press/v28/sutskever13.html

Appendix A
Error and ReLU plots

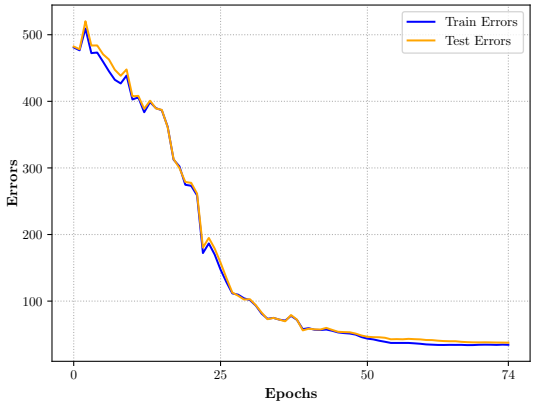

Fig. A1. Training and test error for our model, averaged over 15 runs, with `Tanh` activation.

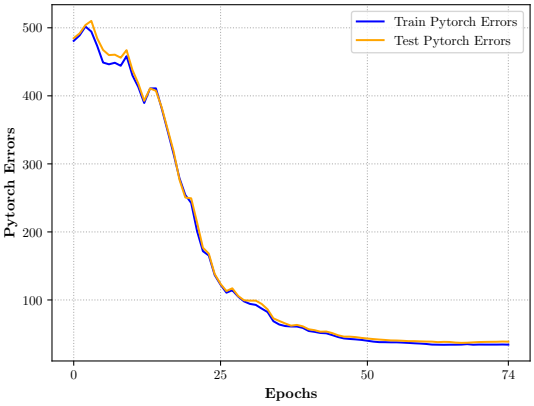

Fig. A2. Training and test error for the Pytorch model, averaged over 15 runs, with `Tanh` activation.

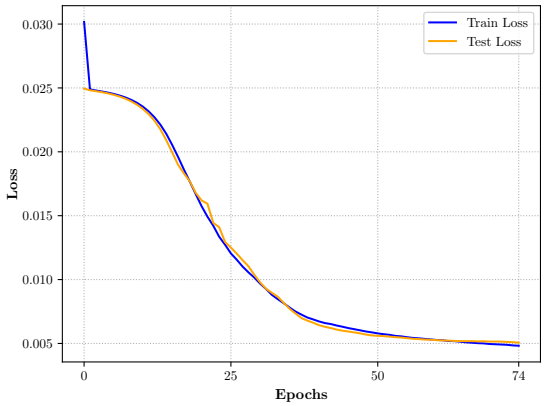

Fig. A3. Training and test loss for our model, averaged over 15 runs, with `ReLU` activation.

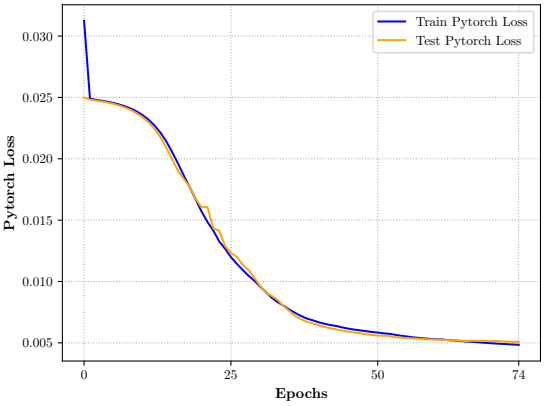

Fig. A4. Training and test loss for the Pytorch model, averaged over 15 runs, with `ReLU` activation.

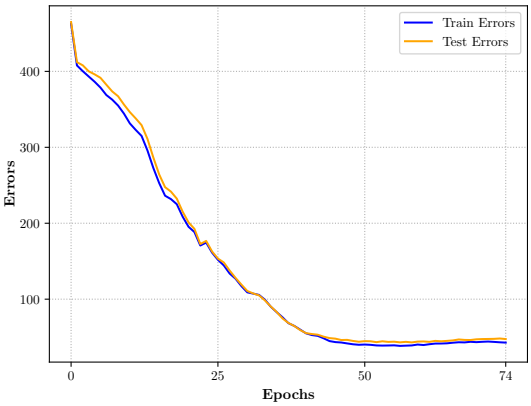

Fig. A5. Training and test error for our model, averaged over 15 runs, with `ReLU` activation.

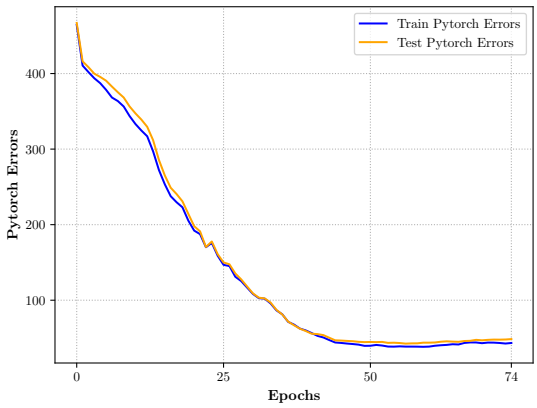

Fig. A6. Training and test error for the Pytorch model, averaged over 15 runs, with `ReLU` activation.