EE559-Deep Learning Project 2 : Mini deep-learning framework

Loïc Vandenberghe - Mathieu Toubeix - Walid Ben Naceur
Department of Electrical Engineering, EPFL, Switzerland

***Abstract*—In this report, we tackle the task of implementing a mini deep-learning framework, using only Pytorch's tensor operations, and the standard math library. We then assess the performance of our framework on a binary classification task, and compare it with Pytorch's on the same task.**

I. Introduction

With the increased interest in deep learning in recent years, there has been an explosion of machine learning tools. Popular frameworks such as Caffe or TensorFlow construct a static dataflow graph that represents the computation and which can then be applied repeatedly to batches of data. Pytorch uses an other approach based on dynamic tensor computations and automatic differentiation.
The aim of this project is to design and implement a basic deep-learning framework, that can instantiate a neural network, and tackle a binary classification task. Our main constraint is to use no pre-existing neural-network python toolbox, hence only using pytorch's tensor operations and the standard math library. The framework must provide the necessary tools to build networks combining fully connected layers, Tanh and ReLU, run the forward and backward passes, and optimize parameters with SGD for MSE.
We also went beyond the minimum requirements of this project, and implemented

- Other loss functions, such as the **L1 Loss** and the **Cross-Entropy Loss**
- Other activation functions, such as the **Sigmoid, Leaky Relu , Tanshrink and Threshold** functions
- The **Dropout** module
- A skeleton for the implementation of Optimizers, so that a developer willing to improve on our framework has a template ready for the implementation of RMSProp or Adam.

We will first describe the implementation of our framework, before analyzing its performance on the binary classification task we were given and comparing it to Pytorch's performance on the same task.

II. Implementing a deep-learning framework

A. Modules

We abstracted the essence of a Module following the suggestion in the project description in the *Module.py* file, where a class following a Module structure implements the following three methods :

```
def forward(self,*input)
def backward(self,*gradwrtoutput)
def param(self)
```

- *forward* defines how the module implements the forward pass of the neural network, it takes tensors as input and return tensors as outputs.
- *backward* defines how the module implements the backward pass of the neural network, it gets as input tensors containing the gradient of the loss with respect to the module's output, accumulate the gradient with respect to the parameters, and return tensors containing the gradient of the loss with respect to the module's input
- *param* returns a list of pairs, each composed of a parameter tensor, and a gradient tensor of same size or an empty list for parameterless modules (e.g. ReLU).

1) Linear Module: This module implements a fully connected layer in a neural network. It follows what is implemented in the torch.nn implementation of fully connected layers. It initializes weights following a normal distribution $\mathcal{N}(0, \sigma^2)$, where $\sigma = 1/\sqrt{n}$ (where n is the number of weights in the layer); and biases to 0.
The *forward* pass simply computes $y = Wx + b$, where W is the weight matrix, b is the vector of biases, x the input features and y the output.
The *backward* pass implements the chain rule, and computes derivatives of the loss function L with respect to the input and to the parameters following the backpropagation equations. We have :

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial W} = \frac{\partial L}{\partial y}X^t$$
$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x} = \frac{\partial L}{\partial y}W$$
$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial b} = \sum_i (\frac{\partial L}{\partial y})_{:,i}$$

We take into account that our training processes mini-batches of samples instead of individual samples during the backpropagation by summing the gradient contributions of independent points.
The *param* method returns the weights and bias of the Module, and the gradients of the loss with respect to them.

2) Tanh Module: This Module implements the non-linear activation function hyperbolic tangent,(tanh) i.e :

$$tanh(x) = \frac{e^x}{e^x + 1}$$

Its derivative is given by :

$$tanh'(x) = 1 - tanh^2(x) = \frac{4}{(e^x + e^{-x})^2}$$

Since the tanh function is already available for tensors, we used the Pytorch implementation of the tanh function. The *forward* pass simply computes the above tanh of the input, the *backward* pass computes the derivative of the loss with respect to the output, namely the derivative of the tanh function defined above. Like all the modules implementing activation or loss functions in our framework, it is parameter less, and its *param* method returns the empty list.

3) ReLU Module: This Module implements the non-linear activation function Rectified Linear Unit,(ReLU) i.e :

$$ReLU(x) = max(0; x)$$

Its "derivative" is defined as :

$$ReLU'(x) = 1_{x>0}$$

The *forward* pass simply computes the above ReLU of the input, the *backward* pass computes the derivative of the loss with respect to the output, namely the derivative of the ReLU function defined above.

4) Sequential Module: This module implements the sequential combination of several modules. We compute the forward pass by computing the forward passes of each sub-module, where the input to the Sequential model is the input to the first sub-module, and the inputs to each sub-module is the output of the previous sub-module of the sequence.
For the backward pass, we iterate on the sub-modules in the reverse order and simply apply the chain rule to compute the gradient of the sequential model.

5) LossMSE Module: We followed the examples available in torch.nn, and considered loss functions as modules. The LossMSE implements the Mean Squared Error loss. It computes the square distance L between the predictions and the targets y, namely :

$$L(\hat{y}, y) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

Depending on textbooks and definitions, many practitioners consider the MSE Loss as the sum of squared errors instead, namely :

$$L(\hat{y}, y) = \sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

Since our goal is actually to minimize this loss function, this does not change the argument which minimizes of the loss, but might have an impact on the choice of a suitable learning rate and the number of iterations needed in the optimization procedure, since the gradients of the loss with respect to y are also scaled by a factor n. What is done is that by default, MSE computes the average loss, however by passing an flag when instantiating this module, it computes instead the sum of squared errors, similarly to what is done in Pytorch.
The derivative of the MSE loss is given by (scaled by n if we use the Sum of squared errors definition instead) :

$$\frac{2}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)$$

The *forward* pass simply computes the above MSE of the prediction and the target (and uses the formula depending on the flag passed at instanciation time), the *backward* pass computes the derivative of the loss with respect to the output, namely the derivative of the MSE function defined above.

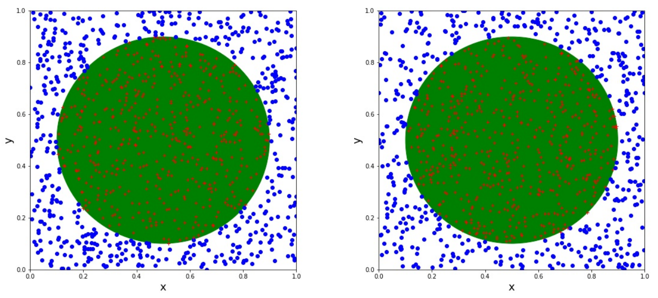
B. *Optimizer Skeleton*

Our main concern while developing this framework was flexibility for the development of further functionalities. In this project, we need to implement stochastic gradient descent in our classification task, we implemented it in an *ad hoc* way as it was sufficient in the scope of this project. However, a developer could be interested in the implementation of other optimization procedures, such as Adam or RMSProp. This is why, following Pytorch's torch.optim, we defined in a similar way to modules a class in *Optimizer.py*. Any optimization procedure in this class should implement a *step* function, that which their parameters depending on the optimization procedure, and a *param* method, similar to the one for Module.

C. *Test.py : Classification task*

We are tasked to generate a training set and a test set of 1,000 points sampled uniformly in $[0, 1]^2$, each with label 0 if outside the disk centered at $(0.5, 0.5)$ of radius $1/\sqrt{2\pi}$, and 1 inside; build a network with two input units,two output units, three hidden layers of 25 units; train it with MSE, log the loss and print the final train and test errors.
To generate the labels, we check whether the L_2 distance between the point $(0.5, 0.5)$ is less than $1/2\pi$ or not, and we observe the following distribution of the data. (see figure [1], blue dots have label 0, red crosses have label 1).

Our network is depicted in figure [2] below :



(a) Train set distribution (b) Test set distribution

Fig. 1: Train and Test set distributions

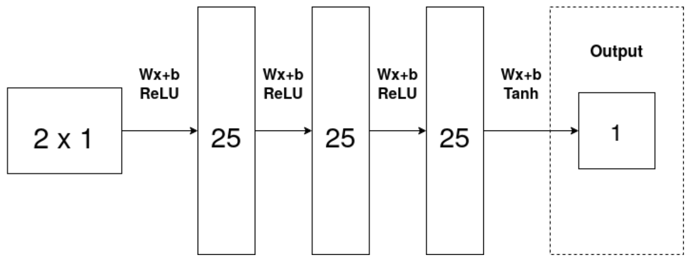


Fig. 2: Architecture of our neural network

D. *Extra Modules*

We decided to implement seven additional modules beyond those that were required

- Two extra loss modules : the **L1Loss** module and the **CrossEntropy** module, following the same pattern as for the MSELoss module
- Four additional non-linear activation functions : the **Sigmoid, Leaky ReLU, Tanshrink** and **Threshold** functions, following the definitions of torch.nn
- The **Dropout** module, which is neither a loss or activation function.By default, some nodes are dropped with probability 0.5 independently, but that probability can be customized in the forward pass. The backward pass returns either the gradients of the loss with respect to the output or 0, depending on the values of the Bernoulli random variables generated in the forward pass.

III. Results

A. *Running our code*

We have ran the code with the following parameters :

Hyper Parameter	**Value**
Learning Rate	5e-1
Batch Size	50
Epochs	100
Criterion	Mean Squared Error

As it is usual to do, we performed data standardization by removing the mean and dividing by the standard deviation of the points. We have run 50 times the training procedure and testing procedure, to obtain an average training error of **3.80 %** and an average test error of **3.60%**. We plot the training error as a function of the number of epochs in figure [3] below :

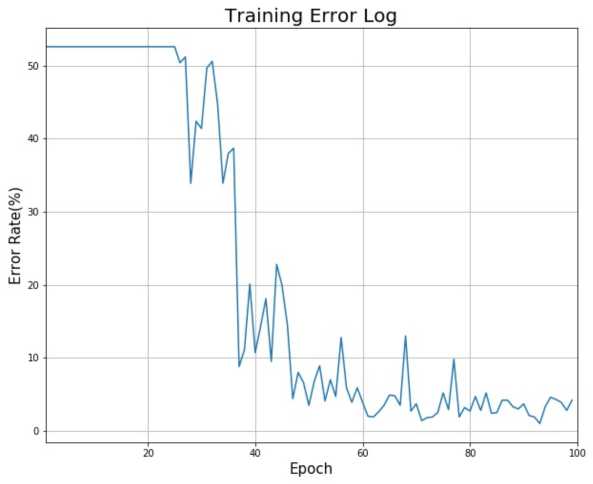


Fig. 3: Train error as a function of the number of epochs

B. *Comparison with torch.nn*

We were interested in evaluating the performance of our model against Pytorch's implementation. This is why we used the exact same parameters (learning rate, batch size, number of epochs, criterion and architecture), and we used the torch.optim procedure for SGD. We have run 50 times the training procedure and testing procedure with Pytorch, to to obtain an average training error of **3.70 %** and an average test error of **3.50%**. We plot the training error as a function of the number of epochs in figure [4] below of our framework and Pytorch :

We can see that, a part from a noisy initialization, we obtain similar performances with Pytorch.

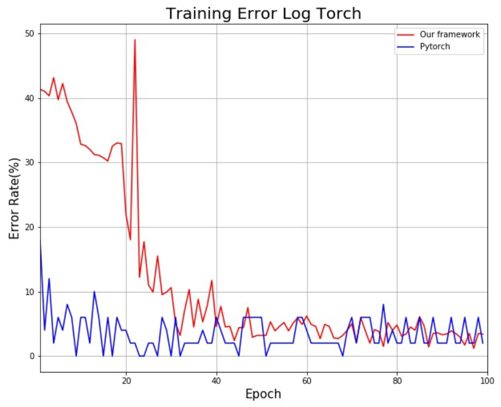


Fig. 4: Train errors of our framework, and Pytorch's, as a function of the number of epochs

IV. Conclusion and discussion

In this report, we have presented the implementation of our mini deep-learning framework. We have described how the several modules we implemented follow a similar pattern, based on a Module skeleton, and described a few bonuses we implemented such as Dropout, and extra losses and non-linear activations. We then described the classification task we had to deal with, and showed that our framework shows similar performance with Pytorch's on this specific task.

One drawback of our implementation however, is that our models must store the data in order to compute the necessary gradients for the learning procedure. Even though, for a task where the dataset is small (1000 points in the Euclidean plane), this is not an issue, as the dataset gets bigger, our solution does not scale.

Many improvements can obviously be made in order to have a fully working deep learning framework, we have only scratched the surface of what is done nowadays with deep learning frameworks : building tools for recurrent neural networks, convolutional neural networks, leveraging GPUs are examples of what could be developed beyond what has been done.

However, throughout this project, we have learned that even though neural networks seems like a complicated mathematical branch of data science, it is possible to build by ourselves in a reasonable amount of time a system that reaches state of the art performance to perform on basic tasks, and that basic neural networks can be easily implemented.

References

- Pytorch's Torch.nn documentation : https://pytorch.org/docs/stable/nn.html
- Pytorch's Torch.optim documentation : https://pytorch.org/docs/stable/optim.html