Project 2 Deep Learning

Victor Dramé
Email: victor.drame@epfl.ch

Arthur Mellinger
Email: athur.mellinger-seron@epfl.ch

Arnaud Guibbert
Email: arnaud.guibbert@epfl.ch

INTRODUCTION

The goal of this project was to implement a deep learning framework, which should in particular enable the user to create and to train a multi-layer perceptron.

In section I., we present the core of our framework, with the different implemented modules and the roles they fulfill. Section II. provides a user guide to create a multi-layer perceptron and to train it on simple task such as recognizing whether a given point is inside a disk in $\mathbb{R}^2$. In section III. we discuss the performances we obtain on this task, together with a performances comparison with PyTorch. We discuss these results in section IV. In this report, all methods, attributes and functions are in **bold**.

I. MODULES

The framework is providing several modules. Some modules are sharing the same parent class and then some general methods. The overall structure of the module classes is detailed in Fig. 1.

A. General methods

We will start with a brief description of the purpose of the main methods and attributes present in all the modules having FrameworkModules as parent (all modules except MSELoss and Optim). Then we define Linear which is a child of FrameworkModules and posses the following method :

- **forward**:
  This method takes as input a tensor of size $N \times D_{in}$ and return a tensor of size $N \times D_{out}$. This method takes as an additional argument a boolean *no_grad* that specifies whether or not you will perform a backward step (i.e. training mode versus evaluation mode). If it is set to False then this methods will append the input to the inputs list (attribute), in order to be able to re-use it during the backward pass. Otherwise the method will not store the output.
- **backward**:
  This method takes as input a tensor of size $N \times D_{out}$ (gradient with respect to the output) and returns a tensor of size $N \times D_{in}$ (gradient with respect to the input). Besides if the module owns parameters to be optimized, then the backward method computes the gradients with respect to these parameters and accumulates them into the gradient attributes (this step will be more detailed in section I-B).
- **reset**:
  This method will reset all the attributes of the module.
- **update**:
  This method takes as input a dictionary where the keys are modules and the associated values are the tensors containing the new parameters. It replaces the parameters of each module by the new parameters tensors. If a module is parameter-less then this method does not modify anything
- **zero_grad**:
  This method resets all the gradients of the model to the zero tensor. If the module is parameter-less then it does not modify anything.
- **params**:
  This method returns a dictionnary with several keys: each key is an object/module and the value associated to this latter is a list of size 2 containing the parameter tensor and its associated gradient tensor. If the object is parameter-less then the list is empty.

As one can notice on the class diagram some specific attributes and classes are provided for each method. These will described in the sections below.

B. Linear

When an instance of Linear is created, the input size D_{in} and the output size D_{out} must be specified. Besides one has to specify whether or not a bias term is added to the output. Then the constructor initializes the weights tensor $W \in \mathbb{R}^{D_{out}\times(D_{in}+1)}$ with Xavier initialization. W is a concatenation of the $\mathbb{R}^{D_{out}\times D_{in}}$ weights matrix and the $\mathbb{R}^{D_{out}}$ biases vector. These parameters are respectively stored in the **weights** attribute. The constructor also initializes the gradient attributes **grdweights** which has the same size as **weights** and stores the gradient of each of its components. The forward step computes $z = XW^\top$ with $X \in \mathbb{R}^{N\times(D_{in}+1)}$ the concatenation of the input $x \in \mathbb{R}^{N\times D_{in}}$ and the vector full of ones. Thus z has size $N \times D_{out}$.

As described in section I-A the module is storing x into the **inputs** list (if *no_grad* is set to False) to be able to re-use it during the backward step. Indeed the backward method of the Linear module is performing the following operation that requires the tensor x:

$$\text{output} = \frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial z} w$$

$$\textbf{grdweights} \mathrel{+}= \frac{\partial \mathcal{L}}{\partial W} = \left(\frac{\partial \mathcal{L}}{\partial z}\right)^\top X$$

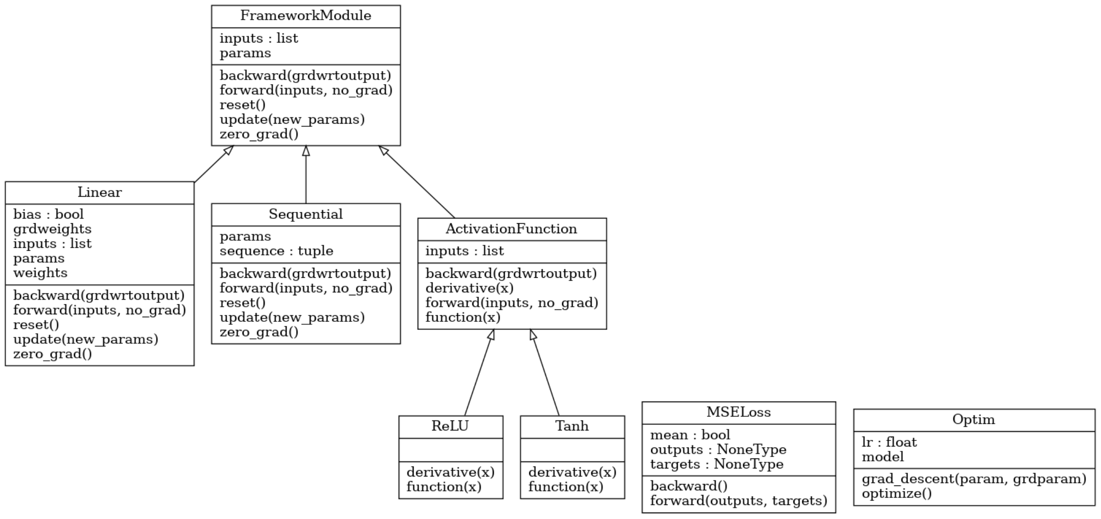


Fig. 1: Class diagram of the framework

with w the D_{in} first columns of W (i.e. W without the biases vector).
That way the module accumulates the gradients into the corresponding attributes, this will allows the user to use mini-batches. It's important to note that, after this operation, the input x is no longer stored into the **input** list of the module. The other methods of Linear are explained in section I-A.

C. Activation functions

A parent class ActivationFunction has been created for all the activation functions. The reason that supports that choice lies in the way the forward pass and the backward pass are computed:

$$\begin{aligned} \textbf{forward} \quad & x = \phi(z) \\ \textbf{backward} \quad & \frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial x} \odot \phi^{'}(z) \end{aligned} \tag{1}$$

Where ϕ is the activation function. As one can notice the only information you need is the activation function ϕ and its first derivative $\phi^{'}$. Therefore one can simply create an activation function module by specifying two methods: **function** and **derivative**. They take as input a tensor and return a tensor of same size corresponding respectively to the function and the first derivative evaluated pointwise. Then the methods **forward** and **backward** (defined in the parent class ActivationFunction) will call these two methods to compute the quantities defined in (1).
The modules Tanh and ReLU has been defined that way. This structure choice increases the modularity of the framework as anyone who wants to add a new activation function can do it simply by defining the two small methods **function** and **derivative**.

D. Sequential

The sequential module takes as input a tuple that defines the module sequence. The Sequential module has the same methods as the ones described in I. When a method is called, the Sequential module will sequentially call this same method for each module. For instance for the **forward** method, it will call the **forward** of the first module and will sequentially iterate until it reaches the last module. For the **backward** method, it proceeds the same way, except that it is done in reversed order.

E. Optim

The module Optim is quite simple and has only two methods and attributes. The constructor takes as input the model to be optimized and the learning rate, they are respectively stored in the attributes **model** and **lr**. The method **grad_descent** is taking as input the parameter tensor and its associated gradient tensor. The function **optimize** extracts the parameters and the corresponding gradients of the model via the **params** method. Then it computes the new weights using the **grad_descent** method and finally pass these new weights to the model via the update method. The new parameters passed to the model is a dictionary where the keys are the modules of the model, and the values are the new parameters tensors.
The choice to create a specific class for the optimization is motivated by the modularity of the Framework. The optimization step could have been hardcoded directly in the modules Sequential or Linear, however this solution would have not allowed flexibility (for instance it would have been impossible to use different optimizers).

II. User Guide (Getting started)

A user guide of the framework is provided there. The framework will be used on a toy data set $\{x_n, y_n\}$ where the classes y_n are either -1 or 1. The data points x_n are sampled in $[0,1] \times [0,1]$ and $y_n = 1$ if x_n is inside the circle of center $[0.5, 0.5]$ and radius $\frac{1}{\sqrt{2\pi}}$, 0 otherwise. A Multilayer Perceptron with three hidden layers, two input units and one output unit will be implemented. The activation function used is ReLU except for the output that is Tanh.

A. Create a model

Creating such a Multilayer Perceptron is quite easy with the Sequential module. It is done using the exact same syntax as with torch.sequential: add instances of Linear, ReLU, Tanh or MSELoss as arguments in the order in which you want the forward path to be computed. As a shortcut, calling the function **create_model** will return the MLP described above

B. Training

First, generate training inputs and targets by calling the **generate_disc_set** function. Training can then be achieved by calling the function **train_model**, which takes as input a model, a train input and a train target, as well as modifiable learning batch. That function computes the forward path using the **forward** method of Sequential, and the loss with respect to the train targets using the **forward** attribute of the criterion. Then resets the gradients to zero using the **zero_grad** method of Sequential; and computes the gradients with respect to the parameters of the model by calling the **backward** methods of criterion and Sequential. The optimization step is realized using the **optimize** method of Optim.

C. Testing

We call again **generate_disc_set** to generate a testing set. Since the two classes to predict are each represented with probability $\frac{1}{2}$, the percentage of error is a good performance measure for the model: it is computed by using **compute_nb_errors** function. Recall that in this testing step one should make sure that the model is in eval mode, i.e. set *no_grad* to True when calling forward method.
To give a sense of how well our framework performs, it is useful to compare it with a state-of-the-art deep learning framework such as Pytorch. To that end, we included the file pytorchNet.py, with a function training the exact same 3-layers perceptron as the one we implemented. Visualisation tools are provided in the file to compare both performances.

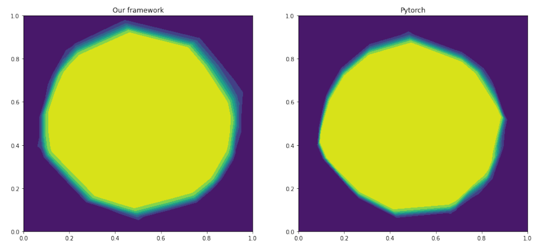
III. Results

A. Accuracy

After training the 3 hidden layers perceptron discussed above over 100 epochs, with mini batches of 50 samples and SGD with learning rate 0.1, we arrive at the following accuracies using the same data sets for our framework and Pytorch, averaged over 10 runs:

Nb of epochs	Our framework	Pytorch
10	85.3	73.1
50	95.6	96.4
200	96.9	97.8

Another way to evaluate the performance is the 2d plot of the decision regions through epochs. This is the kind of results we get after 100 epochs:



Decision regions with (1) our framework (2) PyTorch

B. Running time

On the same CPU, the 3-hidden layers perceptron we discussed is about two times slower to train with our framework than with PyTorch. The average training time per epoch is 0.159s with our framework, and 0.077s with PyTorch.

IV. Discussion

A. Comparison with PyTorch

Our model framework performs about as well as PyTorch on this simple classification task with a MLP. This is explained by the fact that the actual operations performed during the forward and backward paths are basically the same. We observed two minor differences:

- our model learns much faster during the first epochs, which is a benefit from a more careful default Xavier initialization: PyTorch initializes parameters uniformly at random in $[-\sqrt{k}, \sqrt{k}]$ with $k = 1/in\_dim$, while we use $k = 6/(in\_dim + out\_dim)$;
- however after a while, it learns slower. The hypothesis we suggest is the lack of handling of badly conditioned numerical operations, for instance operations involving very small gradient values.