Implementation of a deep learning mini-framework with python

Ridha Chahed, Ghassen Karray, Haitham Hammami
EPFL EE-559 – DEEP LEARNING

***Abstract*— Implementing modern deep learning models can hardly be done without a proper framework to minimize code duplication and maximise ease of use as well as good structure. In this project, we implemented our own mini-framework and compared our performance with pyTorch's NN library, from which it was inspired.**

I. Introduction

The objective of this project is to build a framework written in python to help us design deep learning models, instantiate and train them according to the state of the art methods and steps to finally be able to use them to make predictions. We begin this task from scratch, without the use of any external library apart from standard **math** library (to handle complex mathematical operations) and **torch.empty** (to handle tensor operations).

II. Data Generation

To evaluate our framework, we set up the task of training a simple neural network to classify two-dimensional objects into one of two classes. For our training dataset, we generated a set of 1000 points sampled uniformly in $[0,1]^2$, each with label 0 if outside the disk of radius $\frac{1}{\sqrt{2\pi}}$ and 1 inside. For our test dataset, we generated a set of 1000 points with the same scheme.

III. Typical use case

We encounter the need to use such a framework in supervised machine learning tasks, where we want to approximate an unknown function $f^* : \mathbb{R}^d \to \mathbb{R}^n$ minimising what is known as the expected risk $R(f) = \mathbb{E}_{X,Y}(l(f, X, Y))$
with $l : \mathscr{F} \times \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}$.

If we have i.i.d training samples, $\mathscr{D} = \{(X_i, Y_i)\}_{i=1}^N$, embedded in matrices $X_{train} \in \mathbb{R}^N \times \mathbb{R}^d$ and $Y_{train} \in \mathbb{R}^N \times \mathbb{R}^n$, we can find an estimate $\hat{f}$ to f^* by minimising what is known as the empirical risk
$\hat{R}(f) = \frac{1}{N}\sum_{i=1}^N (l(f, X_i, Y_i)) = L(\hat{Y}_i, Y)$

At this point, an artificial neural network can be seen as a black box, with parameters $\theta \in \mathbb{R}^D$, that computes $\hat{f}(X|\theta) = \hat{Y}$ ($X \in \mathbb{R}^d$, $\hat{Y} \in \mathbb{R}^n$).
The process of deep learning consists of beginning with a certain combination of parameters θ_0 and go through all training samples by groups of B, N_e times, updating θ at each step following a specific update rule giving by an equation of the form $\theta_{t+1} = U(\theta_t, \nabla_\theta L(\theta))$ until we approach a good performance (with $\nabla_\theta L(\theta)$ being the gradient of the loss with respect to parameters).

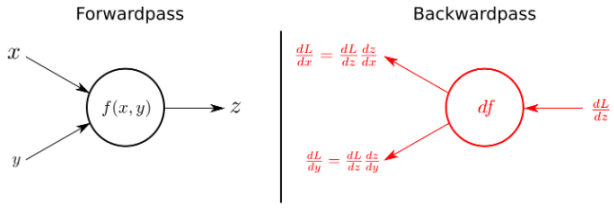

***Fig. 1:** Illustration of forward and backward passes* **[1]**

We implement the process of back propagation by dividing a training step into two substeps, the forward pass, at the end of which we get $\hat{Y}$ and the backward pass, at the end of which we get the totality of $\nabla L(\theta)$ as illustrated in figure 1.

IV. Framework

The main base class around of which revolves the framework is the Module class. One whole neural network can be seen of as an ordered collection of such modules. We defined two other base classes, Initializer to handle parameter initializations and Optimizer to handle parameter updates.

A. Module

The Module class is a base class that forces the classes that inherits from it to redefine three main methods (forward, backward, param). The forward method deals with activations, whereas the backward method deals with gradients. To implement backpropagation efficiently, we made sure to store any useful information

as a field to the module in the forward pass, and use it accordingly to update the gradients with respect to the parameters, also stored as fields, in the backward pass. We make use of the gradients that we stored later in the optimizer step.

B. Containers

We define a container as a module that stores a collection of modules organized in a certain way.

1) Sequential:

- **Instantiation parameters** : *modules

The Sequential organizes the K modules passed to it as argument in a sequential (ordered) manner.

- **forward**($X \in \mathbb{R}^{dim_{in}^{(1)}}$) : $\in \mathbb{R}^{dim_{out}^{(K)}}$
  iterates over all the modules of the collection, in an ascending order, applying the forward function of the current module on the output from the previous module.

$$f(X) = \hat{f}(X) = f_K \circ f_{K-1} \circ ... \circ f_1(X) \quad (1)$$

  with f_i being the result of the forward method of module number i.
- **backward**($\frac{\partial L}{\partial \hat{Y}} \in \mathbb{R}^{dim_{out}^{(K)}}$) : $\in \mathbb{R}^{dim_{int}^{(1)}}$
  iterates over all the modules of the collection, in a reversed order, applying the backward function of the current module on the output from the previous module. It yields the gradient of the loss with respect to the parameters.

$$bp(\frac{\partial L}{\partial \hat{Y}}) = \frac{\partial L}{\partial X} = bp_1 \circ bp_2 \circ ... \circ bp_K(\frac{\partial L}{\partial \hat{Y}}) \quad (2)$$

  with b_i being the result of the the backward method of module number i.

C. Layers

A layer module has learnable parameters. Its forward method depends on these parameters that we initialize using Xavier initialization in order to avoid the layer activation outputs from exploding or vanishing.
In the backward method we store the gradient for these parameters

1) Fully connected linear layer:

- **Instantiation parameters** : dim_{in}, dim_{out}
- **Learnable parameters** : $W \in \mathbb{R}^{dim_{out}} \times \mathbb{R}^{dim_{in}}$, $b \in \mathbb{R}^{dim_{out}}$
- **forward**($X \in \mathbb{R}^{dim_{in}}$) : $\in \mathbb{R}^{dim_{out}}$

$$f_i(X) = XW^T + b$$

- **backward**($\frac{\partial L}{\partial \hat{Y}} \in \mathbb{R}^{dim_{out}}$) : $\in \mathbb{R}^{dim_{int}}$

$$bp_i(\frac{\partial L}{\partial Y}) = \frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y}W$$

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial Y}^T X \qquad \frac{\partial L}{\partial b} = \sum_{i=1}^{N}(\frac{\partial L}{\partial y_i})$$

D. Activation functions

An activation module is a type of module that does not store a gradient and thus does not learn.
It is typically used after Layer modules, so its forward correspond to f_is with even i while it's with the odd i for the layer module, the backward being the other way around.
In activation modules, $dim_{in} = dim_{out}$ meaning the input and output sizes of the transformations are the same. The activation functions we implemented can be found in table I.

Activations	forward	backward
ReLU	$max(0, X)$	$1[x \geq 0]$
LeakyReLU	$max(0, X)+$ $\alpha min(0, X)$	$1[x>0]$ + $\alpha 1[x \leq 0]$
Sigmoid	$\frac{1}{1+e^{-x}}$	$\sigma(x)(1-\sigma(x))$
Tanh	$\frac{e^x-e^{-x}}{e^x+e^{-x}}$	$\frac{4}{(e^x+e^{-x})^2}$

TABLE I: *Activation functions*

E. Losses

- **Instantiation parameters** : network

The loss is a metric we use to model how wrong we are in approximating f^*. It is a function computed on the output of the whole neural network. The losses we implemented can be found in table II.

- **forward**($\hat{Y} \in \mathbb{R}^n$, $Y \in \mathbb{R}^n$) : $\in \mathbb{R}$
  In the forward pass, we store the prediction $\hat{Y}$ and actual label Y as fields and compute $L(\hat{Y}, Y)$.
- **backward**() : void
  In the backward pass, we use $\hat{Y}$ and Y stored in the forward pass to compute $\frac{\partial L}{\partial \hat{Y}}$ and pass it as argument to the backward function of the container stored at instantiation, beginning the computation of gradients chain (backpropagation).

Losses	forward	backward
MSE	$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$	$2(\hat{Y} - Y)$
BCE	$-Y \log \hat{Y} - (1-Y)\log(1-\hat{Y})$	$\frac{\hat{Y}-Y}{\hat{Y}-\hat{Y}\times\hat{Y}}$

TABLE II: *Loss functions*

F. Optimizers

Gradient descent**[2]** is used to to minimize the loss function $L(\theta)$ with $\theta \in \mathbb{R}^D$ the model's parameters by updating it in the opposite direction of the objective function's gradient. Nonetheless, on top on this, we can find various algorithms to optimize this approach in order to obtain fast convergence while avoiding being trapped in a suboptimal minimum.

1) SGD: We can find variants of gradient descent that differ by the amount of data used to compute the objective function's gradient. We have here a trade off between accuracy and computation time. In stochastic gradient descent, the updates are done for every training input x^i and label y^i:

$$\theta = \theta - \eta \nabla_\theta L(\theta; x^i, y^i) \tag{3}$$

In a more general setting we have the mini-batch version that computes it using batch of n samples:

$$\theta = \theta - \eta \nabla_\theta L(\theta; x^{i:i+n}, y^{i:+n}) \tag{4}$$

2) SGD with momentum: One main problem of gradient descent is pathological curvature, that is regions where the loss function isn't scaled in the same way depending on the dimension. While progress can be made in certain directions, it can start oscillating or even grind to a halt along others. Momentum tackles this problem by giving to gradient descent a short-term memory, allowing it to accelerate or reduce oscillations in the relevant directions when needed.

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla_\theta L(\theta) \\ \theta &= \theta - v_t \end{aligned} \tag{5}$$

3) RMSprop: Root Mean Square Propogation aims also to dampen the oscillations but in a different way. It takes into consideration gradient of the past steps by computing the exponential average of their squared value. It then uses it to adapt the parameter using a learning rate inversely proportional to it.

$$\begin{aligned} g_t &= \gamma g_{t-1} + (1-\gamma) \nabla_\theta L(\theta)^2 \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{g_t + \epsilon}} \nabla_\theta L(\theta) \end{aligned} \tag{6}$$

4) Adam: We can consider Adaptive Moment Estimation as a combination of SGDmomentum and RMSprop as it computes the exponentially decaying average of both the past gradients m_t and the past squared gradients g_t.

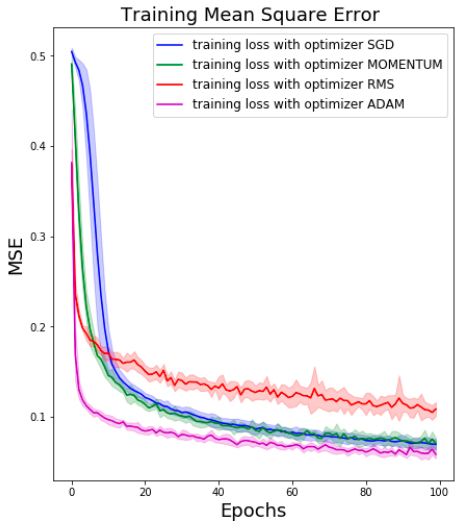

Fig. 2: *Optimizer comparaison with different optimizers*

V. Performance

To evaluate our framework, we set up a basic neural network and trained it with the dataset from section 2. We instantiated a neural network with the same parameters but with pyTorch's NN library, and we compared the results in table III

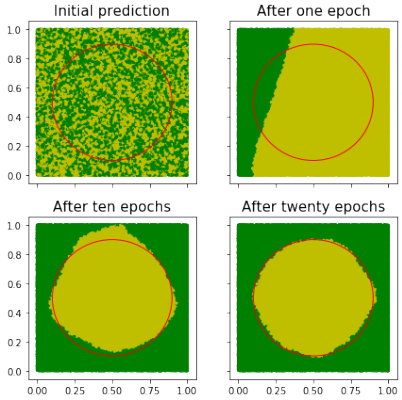

Fig. 3: *Decision boundaries*

Framework	Accuracy	Time
pyTorch	936.5	340.5
Ours	931.4	321.79

TABLE III: *Table to test captions and labels*

VI. Conclusion

The task of building a deep learning framework from scratch turned out to be a very fruitful exercice from which we learned several good practices we will certainly apply in our next pyTorch projects.

REFERENCES

[1] Understanding the backward pass through Batch Normalization Layer, Frederik Kratzert

[2] An overview of gradient descent optimization algorithms, Sebastian Ruder, 2016, arXiv:1609.04747