Deep Learning - Mini Project 2

Sylvain Lugeon, Tyler Benkley, Frédéric Berdoz
Deep Learning, EPFL, 2020

***Abstract*—In this project, we develop a library for neural network implementation. Using PyTorch as inspiration, classes like Linear, Relu, MSELoss are put together such that their functionality mimics that of PyTorch. Testing on a simple classification task revealed similar training results for both our library and PyTorch with the only noticeable drawback being that ours is slower.**

I. Introduction

In this project, we are tasked with building a deep learning framework without the use of `autograd` (automatic differentiation) from PyTorch. The framework must at least enable building sequential models with linear fully connected layers and activation functions such as ReLU (rectified linear unit), sigmoid and hyperbolic tangent. In addition, the framework must provide the means to train such models using stochastic gradient descent (SGD) with a mean square error (MSE) loss. We start off by presenting the library and its constituents and provide a rationale in the next part. Finally, we test our library using a generated test and train data set (both of size 1000 samples) representing points on the two dimensional surface $[0, 1] \times [0, 1]$. The points have a target 1 if inside the disk of radius $\frac{1}{\sqrt{2\pi}}$ centered in $(0.5, 0.5)$, and 0 if outside.

II. Framework

In order to develop a framework that is both practical and intuitive, we modeled it after the well-known PyTorch library `torch.nn`, i.e. a library of modules that are trained using the following procedure:

- A forward pass of an input batch through the model and the loss, storing the internal state in each layer. The batches are `torch` tensors whose samples are aligned along the first dimension and channels along the second.
- A reinitialization (to zero) of the gradient of the loss with respect to the parameters of the model.
- A backward pass through the model, storing the gradient of the loss with respect to the parameters of each layer.
- An optimization step using an optimizer wrapper.

The main challenge in this project arises from the fact that we do not have access to `autograd` in the backward pass. We solved this problem by providing a backward method that can compute and store the gradient in each module representing a neural network layer (linear layers, activations, losses, etc.). The gradients and parameters are stored in `torch` tensors in order to exploit their efficient tensor operations.

III. Architecture

The architecture of our library is depicted in the appendix (Fig. 2 presents the class hierarchy of our framework). We consider an abstract class to be a class who has at least one method that is not implemented, i.e. an attempted use of this method will result in a NotImplementedError. Any such method is said to be a virtual method and bears a (*) in this section. It must therefore be overridden accordingly in sub-classes.

A. Module Class (Abstract)

It is the parent class of all modules of the project. We consider a *module* to be any object that the input of a neural network must go through, either during training or testing. This class contains the following methods:

- *`forward(*input)`: Evaluates the forward function at the given input and returns the result (different for losses, see paragraph III-B). Stores the internal states if needed.
- *`backward(*gradwrtoutput)`: Given the derivative of the loss with respect to the output of the module, updates the gradient of the loss with respect to its parameters (if any) and returns the derivative of the loss with respect to the output of the previous module (different for losses, see paragraph III-B).
- `param()`: Returns an empty list. Must be overridden in modules that have parameters in order to return a list of the parameters of that module. As said above, parameters are stored in `torch` tensors.
- `gradwrtparam()`: Returns an empty list. Must also be overridden in modules that have parameters in order to return a list of gradients of that module (the order of the gradients in the list must match the order of the parameters returned by the method `param`). Gradients are also stored in `torch` tensors.
- *`to_string()`: Returns a textual representation of the module.

B. Loss Classes

Any sub-class of *Module* that represents a loss function. Overrides the forward and backward methods as follows:

- `forward(prediction, target)`: Returns the loss between the prediction and the target.
- `backward(prediction, target)`: Returns the derivative of the loss between the prediction and the target.

The mean square error loss (MSE) and the cross-entropy loss have been implemented.

C. Cell Class (Abstract)

Sub-class of *Module* whose attribute `in_value` stores in memory (if attribute `train` is set to true) the output corresponding to the last input that has been forwarded through. Contains the following methods:

- `training_mode(bool)`: Sets attribute `train` to `bool`.
- `update_in_value(x)`: Sets `in_value` to `x` if attribute `train` is true.

D. Activation Class (Abstract)

Sub-class of *Cell* and parent-class of all activation functions. It implements the following method:

- `*derivative(v)`: Returns the derivative of the activation function applied to the input `v`.

It also overrides the method `backward` inherited from *Module*, which has the same expression for all activation functions given their derivative. The following activation functions have been implemented as sub-classes of the *Activation* class:

- Sigmoid: $f(x) = \frac{1}{1+e^{-x}}$
- Relu: $f(x) = \max\{0, x\}$
- Tanh: $f(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$
- Softmax: $f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$

E. Linear Class

Sub-class of *Cell*, represents a linear layer of a neural network. It is characterized by the attributes:

- `in_size`: dimension of an input sample.
- `out_size`: dimension of an output sample.
- `w`: values of the multiplicative weights of this layer, tensor of size `out_size`×`in_size`.
- `dw`: accumulated gradients of the loss function with respect to each element in `w`.
- `bias`: Boolean specifying if bias must be used.
- `b` (if `bias` is true): values of the additive weights of this layer, tensor of size `out_size`.
- `db` (if `bias` is true): accumulated gradients of the loss function with respect to each element in `b`.

F. Sequential Class

Sub-class of *Module*, forms a complete neural network with the attribute `layers`, which contains the list of layers forming the network, from entry to output. Although it is not a sub-class of *Cell*, it also implements the method `training_mode` such that it successively calls it in each layer forming the network. The class also defines two additional methods:

- `add_layers(*modules)`: Adds a (list of) module(s) at the end of the model.
- `predict(input)`: Forwards the input through the model without storing the internal state in each layer (`train` set to false), and returns the output.
- `describe()`: Prints a textual description of the full neural network.

G. Optimizer Class (Abstract)

It is the parent class of all optimizers and has the attributes `model` which stores the model to be trained, and `lr` which stores the initial learning rate for the optimization. Moreover, it contains two methods:

- `zero_grad()`: Sets all the gradients of the model to zero.
- `*step()`: Performs the optimization step.

The stochastic gradient descent (SGD) and the adaptive moment estimator (Adam) optimizers have been implemented as sub-classes of the *Optimizer* class.

IV. Performance

As presented in Table I (where μ represents the mean and σ the standard deviation), our framework performs well compared to the PyTorch library given the relatively low complexity of our code. There is however room for improvement as far as run-time. In fact, our library is approximately 50% slower than the PyTorch library. This probably stems from the fact that our way of computing and gradients is inefficient compared to the thoroughly optimized `autograd`.

Table I: Performance over 50 Rounds of 25 Epochs

Library	**Test Accuracy**		**Training Time**	
	μ [%]	σ [%]	μ [s]	σ [s]
Own	92.62	2.99	2.20	0.23
PyTorch	93.61	2.57	1.47	0.25

Nonetheless, the test accuracy of a model trained with our framework is satisfactory as seen in Fig. 1. Indeed, a visual difference in test performance between PyTorch and our library is barely visible with no apparent discrepancies in the type of errors. This is an indicator of correctness of our code.

V. Discussion

A. Modularity

The classes *Module*, *Activation*, *Cell* and *Optimizer* serve as abstract classes/interfaces and are not thought to be instanciable. For the simplicity of the code and to follow the suggested structure, we choose not to use the *ABC*

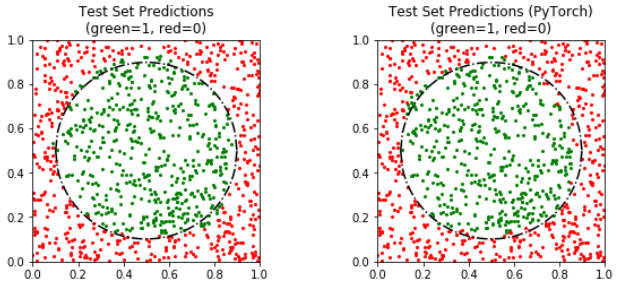

Figure 1: Test predictions using our library (left) and PyTorch library (right) on the same dataset (25 epochs of training). The dashed circles represent the ground-truth boundary.

python module but instead, have all their (considered virtual) methods throw a *NotImplementedError*. This ensures that any sub-module requires these methods to be implemented in order to be callable. The modularity of our code is also improved by the fact that we have abstract sub-classes dedicated to different types of modules. For instance, if one wants to create a new optimizer, it is sufficient to create a sub-class of *Optimizer* and to implement accordingly all the virtual methods inherited from the parent class.

B. Scalability

In addition, we note that the structure of our code allows the imbrication of *Sequential* modules, e.g a *Sequential* module whose layers are also *Sequential* modules. In a general manner, the structure is thought to be easily scalable.

C. Improvement

As mentioned above, one could certainly improve the way we replace `autograd` in our framework, through the use of *hooks* for example. One could also increase the use of *in-place* operations, as it decreases the number of memory operations. It would also be good to pay more attention to numerical inaccuracies, in particular in the activation functions where exponentials and logarithms are used. Finally, in order to improve the user experience and facilitate debugging, one could throw more exceptions.

VI. Summary

All in all, we created a hierarchically-structured tree of classes and sub-classes such that any redundancies would be exploited and provided an existing framework for the easy appending of any loss, activation function or layer type in the context of a neural network. Albeit slower, our library's performance is on par with PyTorch in a simple classification task which is a form of validation.

VII. Appendix

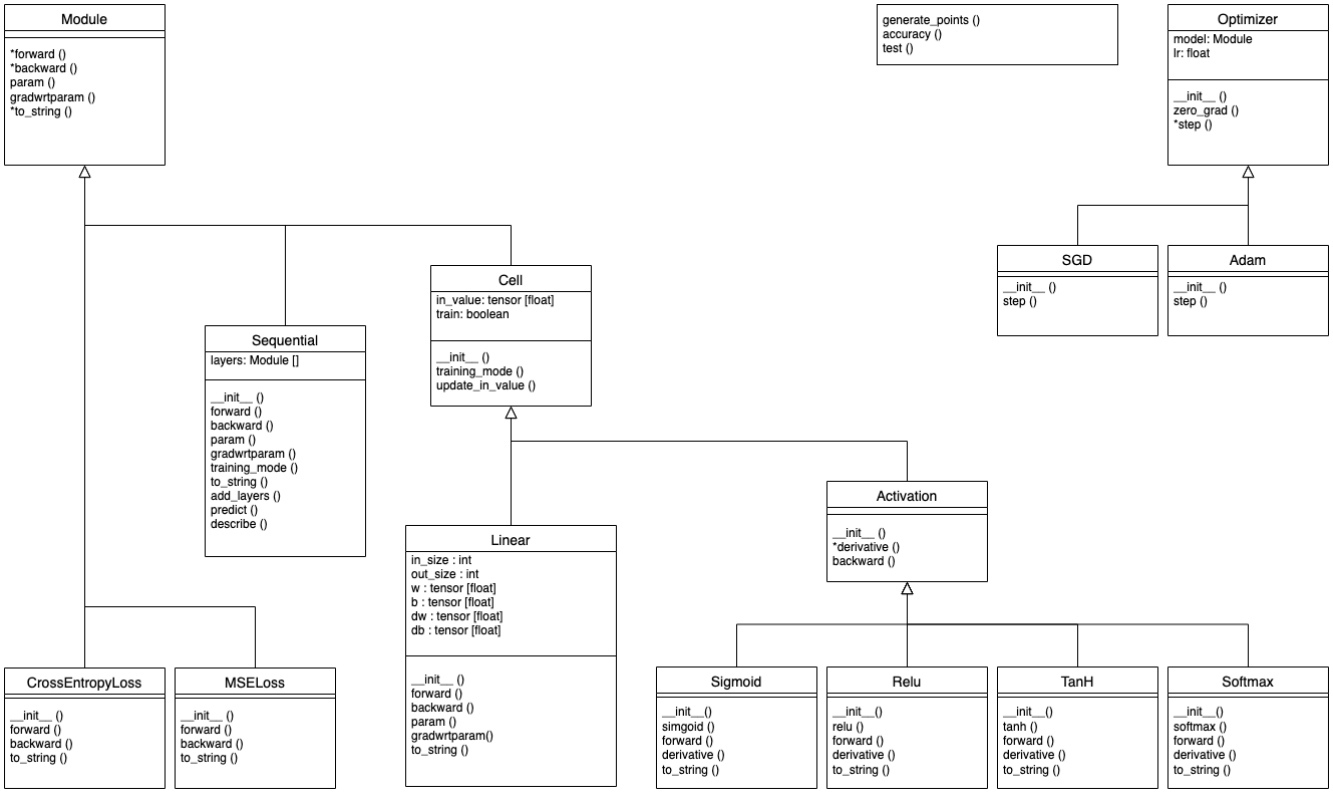

Figure 2: Diagram of the classes, the methods denoted a * raise an error by default and must be overridden.