Design & implementation of a mini deep-learning framework

Nicolas Jomeau, Ella Rajaonson, Sofia Dandjee

***Abstract*— This paper describes the implementation of a mini deep-learning framework which contains the following modules: ActivationFunction (Tanh, ReLU, Sigmoid, Leaky ReLU and ELU), Linear, Sequential, Optimizer (SGD) and Loss (MSELoss, MAELoss and CrossEntropyLoss). The proposed package allows to train a model with fully connected layers as efficiently as the Pytorch library on the classification problem proposed. Its conception makes it easily extensible and maintainable.**

I. INTRODUCTION

Pytorch is a flexible deep learning research platform which provides elegantly designed modules and classes torch.nn and torch.optim, allowing to easily create and train neural networks. However, these easy to use modules can sometimes obscure the understanding of the internal process by the user. The aim of this project was to design a mini "deep learning framework" using only Pytorch's tensor operations and the standard math library to better comprehend the internal machinery behind autograd or the neural-network modules. The framework created only makes use of torch.empty and uses no pre-existing neural-network python toolbox. It provides the necessary tools to build fully connected layers with several activation functions and minimises the mean square error (MSE), mean absolute error (MAE) or the cross entropy loss via stochastic gradient descent (SGD). The correctness of the custom implementation was tested by comparing its performance on a binary classification task of uniform samples points with Pytorch.

II. FRAMEWORK DESIGN

The proposed framework contains the following tools, each implemented using a class or a super-class structure:

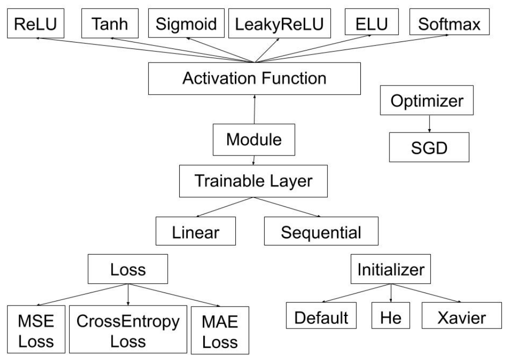

Fig. 1: Architecture of the framework

Super-classes were used to help users extend the API if they weren't satisfied with it. As an example, one could add a new activation layer with the super-class *ActivationFunc* and be sure that it would be correctly used in a *Sequential* model.

A. *Module structure*

The main modules composing the framework were implemented using a general class structure, as followed:

```
class Module(object):

    def forward(self, x):
    def backward(self, *output_grad):
    def requires_grad(self):
    def zero_grad(self):
    def param(self):
```

At this stage only the forward and zero_grad function are defined. They store the input data (that will be later used in the backward pass) and reset the input data to 0, respectively.

B. *Activation functions*

The activation function, which is a *Module*, was implemented using a super-class structure named *ActivationFunc*:

1) The 'forward' function returns the layer's output if given some input.
2) The 'backward' function computes the layer's gradients given the successor layers' gradient.
3) The 'requires_grad' function indicates whether the gradients need to be computed for a given parameter. This function always returns false for the activation functions.
4) The 'param' function returns the layer's parameter in a list. In the case of activation function, this list is kept empty as they are parameter-less.

Several activation functions were implemented as sub-classes; the 'forward' and 'backward' functions depends on the activation function's expressions, and thus constitute the classes' content. The framework proposed contains the following activation functions.

- Sigmoid

$$f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$$

- TanH

$$f(x) = tanh(x)$$

- ReLU

$$f(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases}$$

- Leaky ReLU

$$f(\alpha, x) = \begin{cases} \alpha.x & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases}$$

- ELU

$$f(\alpha, x) = \begin{cases} \alpha(e^x - 1) & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases}$$

The super-class/class format allows the simple implementation of additional activation functions, as it only requires to implement their mathematical expressions (in 'forward') and their derivatives (in 'backward') in an additional class.

C. Trainable Layer

The *Trainable Layer* class is a *Module* whose requires_grad function always returns True.

D. Linear layer

The *Linear* class is a *Trainable Layer* and represents a fully connected layer. It is defined as such:

1) The '__init __' function which initialises the weights and biases to a tensor of given size (in, out) and $(1, out)$ respectively, according to a normal distribution with a standard deviation of 1 by default. Weights can also be initialised according to the Xavier or He initialization, with a standard deviation of $\sigma = \sqrt{\frac{1}{in+out}}$ or $\sigma = \sqrt{\frac{2}{in}}$ respectively. These different initialisations were implemented via an 'Initializer' class.
2) The 'forward' function provides storage of the inputted data for the backward pass computation and returns the layer's output.
3) The 'backward' function stores the next layer's gradient.
4) The 'zero_grad' function, which allows to set the stored gradients or tensors used to compute these gradients to zero.
5) The 'param' function returns a list of pairs for the weights and biases, each composed of the parameter's tensor and gradient tensor.

E. Sequential container

The framework provided also allows to combine several linear layers and activation functions together using the *Sequential* class which is a *Trainable Layer*. It is defined as such:

1) The '__init __' constructs a model with a series of linear and activation function layers.
2) The 'forward' function computes the forward pass on the whole sequential model by going through all the layers of the implemented model.
3) The 'backward' function computes the backward pass on the whole model by iterating from the last layer to first layer. Each layer's gradient is then prepended to the gradient list.
4) The 'param' function returns the parameter's list of each layer crossed.

F. Loss

The loss was implemented as a super-class *Loss*, allowing for the easy implementation of several loss functions. The framework provides built-in implementation of the mean square error (MSE), mean absolute error (MAE) and cross entropy losses. The sub-classes *MSELoss*, *MAELoss*, *CrossEntropyLoss* and their derivatives were implemented using a class structure comprising 2 functions.

1) The 'loss' function returns the computed expression of the loss, averaged.
2) The 'dloss' function returns the computed derivative expression of the loss, normalized with respect to the mini-batch size.

The MAE loss was implemented for its better resistance to outliers than MSE. The cross entropy loss was implemented as it fixes the inconsistencies of the MSE loss in classification problems. MSE is a geometrical measure, which conceptually does not match with classification and it tends to penalize responses that are “too strongly on the right side".

G. Optimizer

Finally, the super class *Optimizer* allows to abstract the update of a model's parameters following a given algorithm. The SGD algorithm was coded as a sub class. It contains 2 functions:

1) The '__init __' constructs the optimizer with the model of which it will update the parameters with a learning rate η. The SGD optimizer also takes a weight decay factor λ as an argument.
2) The 'step' function iterates through the layers of the model, check if they require a parameter update (with the 'requires _grad' function) and updates each of the parameter (for a linear layer, weight and bias) as followed :

$$\Theta_{t+1} = \Theta_t - \eta \nabla \mathcal{L}_\Theta - \lambda \Theta \quad (1)$$

III. Testing

A. Protocol

The performances of the framework were assessed by a classification test on 1000 uniformly sampled points in $[0, 1]^2$ which were labeled as 1 if comprised inside the disk centered at $(0.5, 0.5)$ of radius $1/\sqrt{2\pi}$, or as 0 otherwise. The labels were converted into one-hot encoded labels. The network was built with two input units, two output units and three hidden layers of 25 units and trained with the MSE loss.

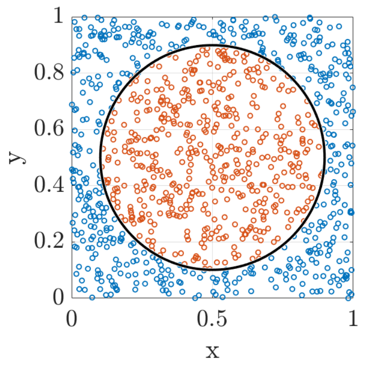

Fig. 2: Visualisation of the classification of the test data. All data points inside the circle (red) are labeled as 1 whereas all data points outside (blue) are labeled as 0.

B. *Results*

The model connected hidden layers with the following activation function (by order of appearance): ReLU, Tanh and Sigmoid. To assess the models' performances, its training and test accuracy as well as its runtime were compared to Pytorch framework's over 10 runs of 1000 epochs each. In both frameworks, the weights were initialized with a standard deviation of 1 and the learning rate was 0.01. The loss evolution over the epochs was also computed and plotted.

With no weight decay (λ = 0):

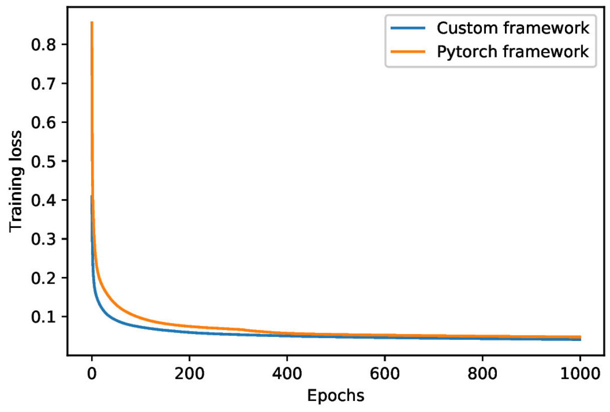

Fig. 3: Comparison of training loss evolution with λ = 0

Architecture	Test acc (%)	Training acc (%)	Runtime
Custom	97.8 ± 0.008	98.2 ± 0.009	14.0
Pytorch	97.5 ± 0.007	98.1 ± 0.006	12.3

TABLE I: Models' training and test accuracy with λ = 0

With some weight decay (λ = 0.001):

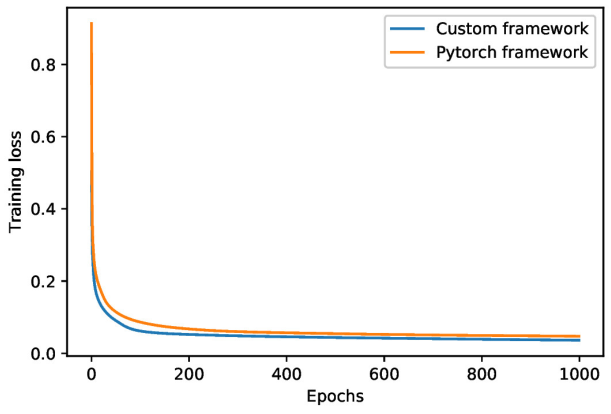

Fig. 4: Comparison of training loss evolution with λ = 0.001

Architecture	Test acc (%)	Training acc (%)	Runtime (s)
Custom	97.9 ± 0.006	98.5 ± 0.006	14.4
Pytorch	97.6 ± 0.008	97.8 ± 0.006	11.6

TABLE II: Models' training and test accuracy with λ = 0.001

C. *Discussion*

As observed on both figures (3, 4), the loss behavior of the proposed framework is comparable to Pytorch's. The weight decay implemented in the SGD optimizer does bring accuracy improvement, but it remains minor due to the already high performances observed. The runtime is about 3s higher using the proposed framework, which leaves room for improvement. We suspect it may be caused by Tensor operations that aren't in-place when they should be in order to avoid Tensors cloning/copy time.

IV. CONCLUSION

The provided framework is able to efficiently solve the classification problem proposed with performances comparable to those of Pytorch's framework, except for the runtime speed. Several improvements could be done to bring some more flexibility to the framework. For example, the latter does not accumulate the inputted data further than the current layer computed and thus cannot implement Siamese networks as it may only retain propagation of one path. Other types of optimizer such as Adam could also be implemented, same for the activation functions and the losses. This could easily be done through the use of the provided super-classes.