Design of a Deep Learning Framework

Hugo Michel
hugo.michel@epfl.ch

Daniel Tadros
daniel.tadros@epfl.ch

Gianni Giusto
gianni.giusto@epfl.ch

Abstract

The current project aims to build a framework from scratch using deep learning algorithms in order to solve basic classification tasks. The obtained results are finally compared to a similar architecture implemented in PyTorch. By developing modular and flexible tools, our framework turned out to be efficient in solving the given task.

1 Introduction

Common Python frameworks such as PyTorch or TensorFlow provide a wide variety of modules and methods to implement artificial neural network architectures. These building blocs aim to perform efficiently tasks requiring the use of deep neural networks but also to ease the conception of such networks. Indeed, by looking closer at the sequences of code, one can easily understand its stepwise execution.

The current project aims at solving a basic classification task by using methods implemented from scratch. Given a set of points, the network determines if the given point belongs to the outside of a circle of radius r (label 0) or the inside (label 1). Only `torch.empty` was used to defined tensors properly together with basic tensor operations from the PyTorch library. No other external Python toolbox were used.

2 Methods

2.1 Classification task

The training and testing sets consist each of 1000 pairs of points drawn from an uniform distribution between $[0,1]^2$. The associated label are 0 if outside a circle of radius $1/\sqrt{2\pi}$ or 1 if inside.

2.2 Framework architecture

2.2.1 Modules

The `Module` class represents the centerpiece of our architecture: specific modules such as activation function modules, inherit from it. The forward and backward passes, which constitute the two main functions of the backpropagation algorithm (Fig. 1), are instantiated in it and redefined in subsequent classes. A simple representation of the architecture is shown in Fig. 2.

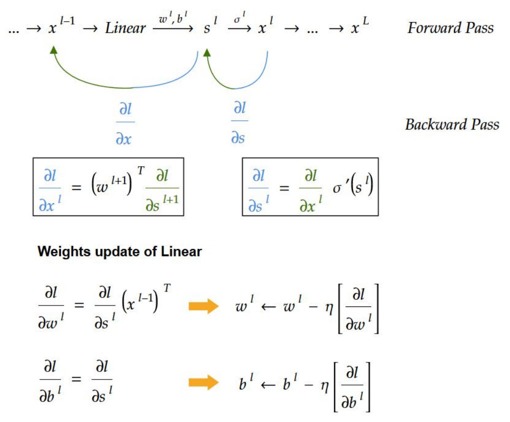
$\ldots \to x^{l-1} \to Linear \xrightarrow{w^l, b^l} s^l \xrightarrow{\sigma^l} x^l \to \ldots \to x^L$
Forward Pass
$\frac{\partial l}{\partial x}$
$\frac{\partial l}{\partial s}$
Backward Pass
$\frac{\partial l}{\partial x^l} = (w^{l+1})^T \frac{\partial l}{\partial s^{l+1}}$
$\frac{\partial l}{\partial s^l} = \frac{\partial l}{\partial x^l} \sigma'(s^l)$
Weights update of Linear
$\frac{\partial l}{\partial w^l} = \frac{\partial l}{\partial s^l} (x^{l-1})^T$
$w^l \leftarrow w^l - \eta \left[\frac{\partial l}{\partial w^l}\right]$
$\frac{\partial l}{\partial b^l} = \frac{\partial l}{\partial s^l}$
$b^l \leftarrow b^l - \eta \left[\frac{\partial l}{\partial b^l}\right]$

Figure 1: Diagram representing the forward and backward passes. For the latter, the inputs to the next layer are represented by *green* arrows whereas blue arrows denote the output of the previous layer. Weights and biases are updated using the gradient obtained after performing the backpropagation (bottom).

Activation function modules

Activation functions are necessary to break the linearity and classify more complex elements. Hence, *Sigmoid*, Rectifier Linear Unit (*ReLU*) and the hyperbolic tangent (*Tanh*) were implemented together with their derivatives in the forward and backward methods of their respective classes.

Linear module

The *linear* module compute the activation of the next layer (a') using the activation of the current layer (a), such that: $a' = \sigma(wa + b)$, where w corresponds to the weight tensor, b the biases and σ the activation which is applied element-wise. The backward pass aims at quantifying how small changes in weights and biases modify the loss. Hence we computed the derivatives of the loss with respect to the activation and finally the derivatives of the loss with respect to the parameters (weights and biases), *i.e.* $\frac{\partial l}{\partial w}$ and $\frac{\partial l}{\partial b}$, as shown in Fig. 1.

Finally, the Xavier uniform distribution[1] has been used to initialize the weights and the biases in the module. This avoids the problem of vanishing gradients, *i.e.* when they become null due to bad parameters initialization.

Sequential module

Sequential constitutes the core module of this framework. Indeed, it contains all other modules and can perform the backward and forward passes throughout the entire network. A simple function enables the user to add or remove a module at the end of the instantiated object *network* and allows an intuitive implementation as in the following example:

```
network.add(Linear(2,25))
network.add(Sigmoid())
network.add(Linear(25,25))
network.add(Sigmoid())
network.add(Linear(25,25))
network.add(Sigmoid())
network.add(Linear(25,2))
network.add(Sigmoid())
```

As the goal of the architecture consists in independent modules that should work separately, the output of the backward and forward is used as the input of the next ones. It allows a maximal flexibility and modularity in the framework.

2.2.2 Optimizers

To make our neural network learn, gradient descent and Adam optimizers were implemented in order to minimize the loss. New weights w and biases b are calculated following the update rule shown in Fig. 1 (bottom) with the help of $\frac{\partial l}{\partial w}$ and $\frac{\partial l}{\partial b}$ computed during the backpropagation algorithm.

[1] Xavier Glorot and Yoshua Bengio, *Understanding the difficulty of training deep feedforward neural networks*

2.2.3 Miscellaneous functions

In addition to the core of the framework described above, some functions were coded as part of this project. Two very important ones are of course the loss and its derivative, which allow to perform the forward and backward passes towards a minima. Then, the computation of the error is also embedded in a specific function that finds the maximal power among two classes (in or out the circle). A *Dropout* module was also implemented to improve the training. Finally, the random training and testing sets of points together with their respective labels come from a generator function coded in a separated file.

2.3 Testing our framework

To test the performance of our *mini*-framework, the losses and the accuracies were recorded throughout the training and testing processes and then compared with a simple fully connected network implemented with `PyTorch`. To allow unbiased comparison, a fully connected network with 2 input units, 3 hidden layers each of size 25 and 2 output units were implemented with the 2 different frameworks. Both were trained by minimizing the mean squared error (MSE). Finally, weights and biases were generated randomly following Xavier initialization. The same *seed* was used to initialize the parameters of the different frameworks. 300 epochs were completed for the training.

3 Results

Our framework displays a 0.0% and 8.6% training and testing error respectively, whereas `PyTorch` performs with 0.6% and 1.10% in training and testing (Table 1). Although we observed a marked difference in testing, the loss across the epochs displays a similar behavior as the one observed with `PyTorch` (Fig. 3). Indeed, an important decrease after few epochs precedes a slow gradual change toward 0. Note that the scale are different for both losses because two different definitions of the MSE loss were used.

Table 1: Training and testing errors

Framework	Train error [%]	Test error [%]
Our framework	0.0	8.6
`PyTorch`	0.6	1.10

Despite a similar accuracy on the training set for

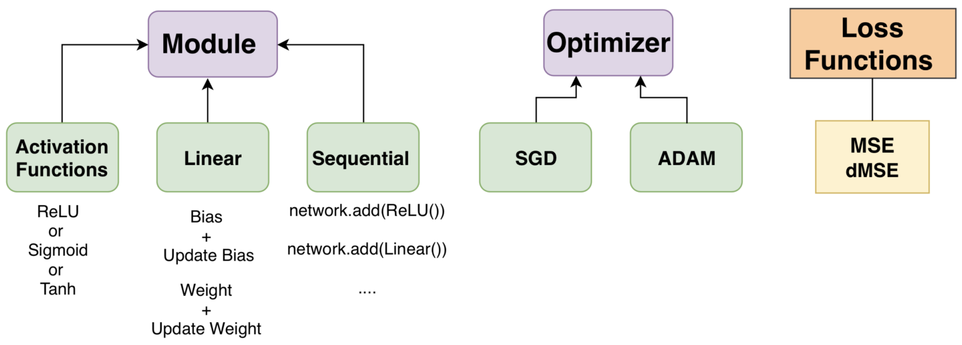

Figure 2: Mini-Framework architecture. *Module* and *Optimizer* represent the main classes (purple boxes). Classes that inherit from the main classes are displayed in shaded green boxes. Inheritance link are represented by black arrows. Loss functions as well as miscellaneous functions are represented in orange.

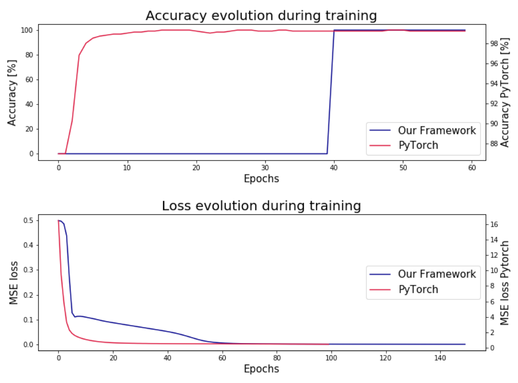

Figure 3: Accuracy and loss evolution during training.

both frameworks at the end, the initial behaviour is quite different. Whereas the PyTorch one increases sharply over 3 to 4 epochs, the one from our framework rises to 100% accuracy in only one epoch (Fig. 3).

4 Discussion

The performance of the 2 frameworks in terms of train and test errors are significantly different. Despite the fact that our framework displays higher testing error rates, the difference between the training and the testing remains small which indicates that no overfitting occurs. The 0.0% training error obtained with our framework may suggest that it is specific to the given task and may fail to generalize. Furthermore, PyTorch functional are optimised and not surprisingly lead to better overall results.

The appropriate behavior of the loss through the learning process proves that optimizers perform well at updating the weights and biases. Our network is hence capable of learning and converges to a minima with suitable parameters.

The choice of this architecture was motivated by its simplicity and modularity. Indeed our framework provides a compact notation to build simple fully-connected layers. For example, the gradients during the backward pass are simply passed to the next layer, without keeping them in memory. Another possible implementation would have been to accumulate the gradient during the backward step and use Python indexing to re-use the appropriate gradient for the computation of $\frac{\partial l}{\partial w}$ and $\frac{\partial l}{\partial b}$ (an example can be found in the book of Michael Nielsen[2]).

5 Conclusion

Our *mini*-deep learning framework turned out to be adapted to this classification task and the generated dataset. Further test should be performed to assess generalization. It could indeed be well suited to this specific task but perform poorly for other classifications. Furthermore, the choice of the parameters could be validated and further optimized through a grid search. In addition, the classification can be improved using the *cross entropy* as a loss function. Indeed, this criterion is often preferred according to the literature in case of classification.

References

Xavier Glorot and Yoshua Bengio. DIRO, Universit de Montreal, Montreal, Quebec, Canada. *Understanding the difficulty of training deep feedforward neural networks*.

[2]http://neuralnetworksanddeeplearning.com/