Mini-project 2: Mini deep-learning framework

Tobia Albergoni, Matteo Yann Feo, Quentin Bouvet

EE-559 Deep Learning - Spring 2019

Introduction

The goal of this report is to present the small deep learning framework developed in the scope of the second course mini-project. The document will go over the structure and content of the code-base, clarifying implementation details and peculiarities. Subsequently, a small *fully connected* network constructed with the framework is instantiated and tested on a non-linear classification task with a toy dataset. The framework is developed with the goal of providing a suite as similar as possible to the PyTorch neural-network toolbox, both from interface and functionality standpoints. Therefore, an equivalent PyTorch model is tested alongside the homemade network in order to provide a confrontation reference for performances and code, and the results of the comparison are presented.

The Framework

Goals and intended result

The project focus is to provide fast and polished implementations of the minimum requirements stated in the handout. Here the scope of the framework is explicitly clarified. It should should be able to:

- Build networks with fully connected layers and a handful of the most common activation functions.
- Use PyTorch tensors as the data structures for inputs, outputs and underlying implementation details.
- Implement the backpropagation algorithm in order to optimize a loss function with Stochastic Gradient Descent optimization.
- Make the implementation complying with a mini-batch SGD implementation.

Structure and content

The functional units are implemented in a modular way, in order to decouple and distribute functionality over appropriately defined classes. Nevertheless, most of the framework modules are **network components** that need to implement the *forward* and *backward* passes as seen in class. Under this category we can find **containers**, **layers** and **activation functions**, and all of them extend a basic *Module* class. Also **loss functions** extend this class and inherit the back-propagation algorithm methods, even if they are not strictly considered as network components. The base *Module* provides the following interface:

forward(input) Implements the forward pass of the module, which takes the input of the network or the output of the preceding layer as input. If necessary, it stores the input and/or output of the last forward call in order to use it during the *backward()* call.

backward(prevGrad) Implements the backward pass of the module by receiving the gradient of the loss with respect to the output of the last forward call and computes the gradient with respect to the inputs, using the results of the last forward pass if applicable. If the module has parameters, it computes the gradient with respect to those parameters and updates internal gradient accumulators, needed during the optimization steps.

zero_grad() Resets the parameter gradient accumulators of the module and of any nested module.

Additionally, the *Module* class provides attributes to store *parameters* and the respective *gradient accumulators*, and an *exception handler* responsible for input validation. Note that the above methods work with tensor shapes that adhere to the $n \times m$ PyTorch convention, where n is the number of samples in the mini-batch currently processed, and m is the dimensionality of one sample.

Finally, the **optimizers** are the only remaining modules that do not extend this class.

Modules and implementation

Layers

The framework implements only fully connected layers, represented by the class *Linear*. This class is instantiated with input dimension i and output dimension j, the latter representing the number of units of the layer. It has two internal parameters: a $j \times i$ *weight matrix* and a $1 \times j$ *bias vector*. As for all classes extending *Module*, the parameters are stored in the dictionary *self.params*, and the identically shaped gradient accumulators in *self.grad*. The gradients with respect to the parameters are computed and accumulated at each *backward* call, while the forward pass simply stores inputs and outputs. All calculations are done according to the formulas presented in class, which are outside of the scope of this report. Note that, according to the principles presented in the course, initialization is 0 for the components of the *bias vector*, while the *Xavier normal scheme* is used for the *weight matrix*, initializing weights with a normal distribution centered at 0 and with standard deviation:

$$std = gain \times \sqrt{\frac{2}{input\_dim \times output\_dim}}$$

Where *gain* is a parameter that depends on the activation function that follows the *Linear* module.

Activation functions

The activation functions currently implemented are simpler components: they have no parameters to learn and thus no internal gradient must be accumulated. The forward pass simply stores the result of the pass, while the backward pass uses it together with the gradient passed as parameter to compute the gradient with respect to the inputs. The following activation functions are implemented:

Rectified Linear Unit (ReLU)

$$f(x) = max(0, x) \qquad \frac{\partial f}{\partial x} = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{otherwise} \end{cases}$$

Hyperbolic Tangent (Tanh)

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \qquad \frac{\partial f}{\partial x} = 1 - f^2(x)$$

Logistic Sigmoid function

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

Containers

Modules need to be chained together to obtain a functioning model, and this is the goal of the container class *Sequential*, which represents a sequence of network components and is responsible of representing a complete model. It is thus the interface used to launch forward and backward passes over the entire net. Upon calling *forward()* on a *Sequential* object with an input, the class will iterate in order over the sequence of components of the net, calling the corresponding *forward()* methods with the output of preceding modules. The *backward()* call mirrors this behavior in opposite sense, propagating the gradients. Lastly, calling *zero_grad()* on this container propagates the call to all contained modules.

Loss functions

Loss functions extend the base *Module* class and are used during the optimization procedure for two purposes: compute the loss given predictions and true targets, and compute the gradient of the implemented loss function with respect to the model's output. Those two behaviors are implemented by the *forward()* and *backward()* methods respectively. The only loss function currently implemented is the **Mean Squared Error (MSE)**, which has the following expressions for a single sample:

$$L(x_n, y_n) = (x_n - y_n)^2 \qquad \frac{\partial L}{\partial x_n} = 2(x_n - y_n)$$

For the loss computation on a mini-batch, the class outputs by default the sum of the sample losses, but there's also the possibility to obtain the mean loss per sample.

Optimizers

Finally, optimizers are simple objects that need only a *step()* method and are instantiated with a model to optimize. The only implementation in the framework uses the **Stochastic Gradient Descent (SGD)** algorithm to update the parameters of all network components after processing each mini-batch of samples. This is done by iterating over all parameters of all modules contained in the network, and adding to them the matching gradient accumulator, scaled by the given learning rate. Usually, the gradient accumulators are reset through *zero_grad()* after each optimization step, but this behavior is external to the *SGD* class.

Performance evaluation

Setup

The suggested non-linear classification task (cfr. handout for details) is used to evaluate the performance of the framework. As anticipated, identically structured networks are created both with the framework and with pure PyTorch. The architecture is the one presented in the handout, with a $ReLU$ activation after each layer, except for the last one.
The evaluation is performed over 10 training runs. Each time, a new dataset of 1000 training/validation samples is generated. The two models are re-initialized and then trained with the following setting: 1000 training epochs, mini-batches of 50 samples and a learning rate of 0.001. During each run, the training and validation losses are recorded at every epoch. Training time is also measured, and the final training and validation classification accuracy is computed.

Results

The results reported in Table 1 show the average validation accuracy. The figures show that the framework's performance is really close to a full PyTorch implementation, both in terms of accuracy and training time. A difference of less than 0.2% is a satisfactory achievement. On top of that, considering that PyTorch uses autograd to compute the gradients, while the framework uses a classical backpropagation implementation, such a small training time difference is noteworthy.

	Test accuracy	**Training time**
Framework	0.9699 ± 0.0093	13.16 s ± 0.75
PyTorch	0.9717 ± 0.0072	12.87 s ± 0.92

Table 1: Mean accuracy and times

Nevertheless, this should not be taken as evidence of the equivalence of the models. First, it should be noted that the task at hand is a simplistic one, and results might not hold for more complex situations. Second, end-of-training results don't tell the whole story, which is the reason for the plots in Figures 1 and 2. They show the training history (loss and accuracy) of the framework and PyTorch models, respectively. The bold lines are the means, while the colored area represents one standard deviation confidence intervals.

Figure 1 shows that the framework needs almost the full extent of the 1000 training epochs to reach the final loss and accuracy results. It can also be observed that after the 500^{th} epoch, the standard deviations of both metrics reduce to small values and hence they converge to similar results in all training runs.

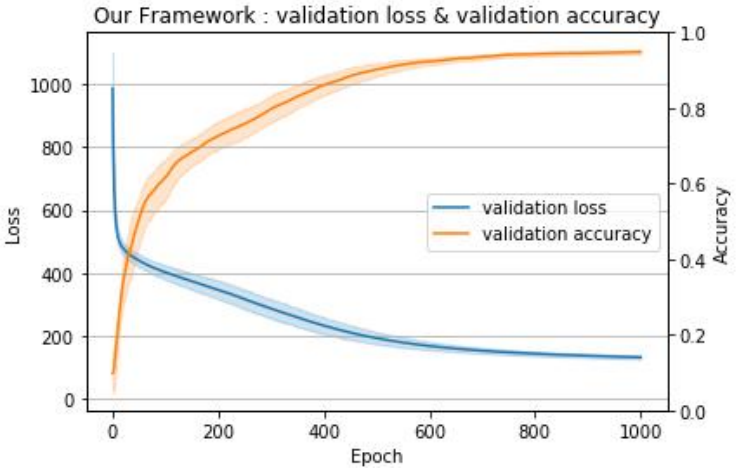

Figure 1: Framework model training history

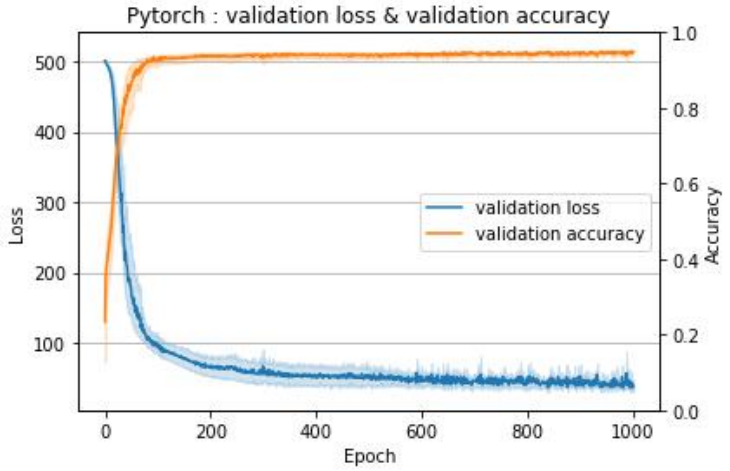

Figure 2: Pytorch model training history

The training history of the PyTorch model can be observed in Figure 2. It is clear that this implementation converges much faster than the framework. Training could be stopped after just 100/200 epochs and the model would produce almost the same results. This analysis of training histories highlights that the PyTorch implementation is as expected superior to the framework, an insight that couldn't be extracted from just the end-of-training results.

The results displayed here are computed with the script *test.py*. However, the plots are generated using a separate Jupyter notebook, *mkplot.ipynb*, which uses the *seaborn* package (unavailable in the VM), attached to the code submission.