Mini Deep Learning Framework

Virginia Bordignon
EE-559 Deep Learning, EPFL, Switzerland

***Abstract*—In this report, we present a Deep Learning framework designed using only tensor operations and the math library in Python. In the framework, different modules such as linear layers, activation functions, loss functions and optimizers were implemented, all of which inherit properties from a fundamental parent module class. These modules can be therefore combined into multilayer perceptrons (MLPs). The functionalities of the framework are illustrated by means of simulated data for a binary classification task.**

I. Introduction

Among the many existing deep learning (DL) frameworks, PyTorch has become widely popular both in research and professional applications. The popularity can be traced back to its tensor-based computations, which leverage GPU processing, and to its fast autograd functionality [1]. The high-level language used in designing DL models is extremely advantageous, which makes their deployment possible even to users with more limited understanding of the underlying operations.

While relying on pre-designed modules is useful, it is valuable for students and researchers to understand the mechanisms behind the PyTorch library. Aiming to improve understanding of such tools, this work proposes to build and present a simplified DL framework, built using tensor and fundamental math operations.

In this report, we discuss the modules implemented and how they can be used to build MLPs for regression or classification taks. We illustrate their use for a binary classification problem which was numerically generated. The use case is a way to compare the different implemented modules and showing their correct operation.

II. Framework Modules

A. Layer Modules

In this framework, we considered linear layers and a sequential container, which allow us to build deep MLPs. At the $\ell-$th linear layer, the following affine combination is implemented in the forward pass:

$$y^{(\ell)}(x) = W^{(\ell)} {x^{(\ell-1)}}^{\mathsf{T}} + b^{(\ell)} \tag{1}$$

with weight matrix $W^{(\ell)}$ and bias vector $b^{(\ell)}$ being the parameters of said layer. The gradients needed for the backward pass are computed as functions of the gradient with respect to the output, i.e., $\left[\!\!\left[\frac{\partial \ell}{\partial y^{(\ell)}}\right]\!\!\right]$. Their expressions can be seen in Table I. The sequential container combines different modules in series, propagating the forward and backward operations across subsequent modules.

Table I
Linear layer and gradient expressions.

	Gradient
$\left[\!\!\left[\dfrac{\partial \ell}{\partial w^{(\ell)}}\right]\!\!\right]$	$\left[\!\!\left[\frac{\partial \ell}{\partial y^{(\ell)}}\right]\!\!\right]^{\mathsf{T}} x^{(\ell-1)}$
$\left[\!\!\left[\dfrac{\partial \ell}{\partial b^{(\ell)}}\right]\!\!\right]$	$\left[\!\!\left[\frac{\partial \ell}{\partial y^{(\ell)}}\right]\!\!\right]^{\mathsf{T}} \mathbb{1}$
$\left[\!\!\left[\dfrac{\partial \ell}{\partial x^{(\ell-1)}}\right]\!\!\right]$	$\left[\!\!\left[\frac{\partial \ell}{\partial y^{(\ell)}}\right]\!\!\right] W^{(\ell)}$

B. Activation Functions

In Table II, we see the three activation functions implemented in the DL framework: ReLU, Sigmoid and Tanh. In the ReLU case, since it is a non-smooth function, we consider its subgradient instead of gradient.

Table II
Activation functions and their gradient expressions.

	Activation Function	**(Sub)Gradient**
ReLU(x)	$\max(x, 0)$	$\mathbb{1}[x > 0]$
Sigmoid(x)	$\dfrac{1}{1+e^{-x}}$	$\dfrac{\text{Sigmoid}(x) - 1}{\text{Sigmoid}(x)}$
Tanh(x)	$\dfrac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$	$1 - \text{Tanh}^2(x)$

C. Losses

To enable both regression and classification tasks, we have chosen to implement the mean-squared error loss and the cross-entropy loss. Consider $\boldsymbol{y}$ as the target value/ batch and $\boldsymbol{x}$ the input sample/ batch. The MSE loss is computed as

$$\mathsf{MSE}(\boldsymbol{x}; f) = \|\boldsymbol{y} - \text{sigmoid}(f(\boldsymbol{x}))\|^2 \tag{2}$$

where $f(\cdot)$ is the function implemented by the MLP. The MSE can be primarily used for regression problems. Its gradient is given by:

$$\mathsf{MSE}'(\boldsymbol{x}; f) = -2(\boldsymbol{y} - f(\boldsymbol{x})). \tag{3}$$

We have also implemented the cross-entropy loss for classification problems with set of classes $\mathcal{C}$:

$$\mathsf{H}(\boldsymbol{x}; f) = -\sum_{k\in\mathcal{C}} \boldsymbol{y}_k \log q_k(\boldsymbol{x}) \tag{4}$$

where the last layer of the MLP goes through a softmax operation:

$$q(\boldsymbol{x}) = \mathrm{softmax}(f(\boldsymbol{x})). \tag{5}$$

Its gradient is given by:

$$\mathsf{H}'(\boldsymbol{x}; f) = q(\boldsymbol{x}) - \boldsymbol{y}. \tag{6}$$

D. Initialization Strategies

The choice of initialization has the main objective of controlling the variance of the gradients w.r.t. the weight and bias arrays. It is known that for deeper networks, the gradient can either explode or vanish (due to some ill conditioning of the weight matrices, for example). One way to control this variance is by introducing some initialization.

In the uniform initialization, the weights are generated according to a uniform distribution $\mathcal{U}[-\frac{1}{\sqrt{N_\ell}}, \frac{1}{\sqrt{N_\ell}}]$, where N_ℓ is the size of the input layer. In the Xavier normal initialization, the weights are generated according to a uniform distribution $\mathcal{N}\left(0, \frac{2}{\sqrt{N_\ell + N_{\ell+1}}}\right)$, where $N_{\ell+1}$ is the size of the output layer.

E. Optimizers

As algorithm to solve the optimization problem, we have implemented stochastic gradient descent (SGD), and the following adaptive gradient methods: RMSProp, Adam, Adagrad according to the algorithm in [2] and Adadelta, according to the original reference [3].

Consider the parameterized loss function $\ell(x; w)$ and its parameter w. Then SGD performs the classical update:

$$w_i = w_{i-1} - \mu g_{i-1} \tag{7}$$

where μ is a fixed learning rate and $g_{i-1} = \frac{\partial \ell(x; w_{i-1})}{\partial w}$. In the adaptive formulations, simultaneously to the update the first and second moments of the gradient variable are estimated using a running average recursion. These algorithms scale the gradient updates using these estimates in different ways.

For example, in Adam [4], the first and second error moments are estimated as:

$$s_i = \rho_1 s_{i-1} + (1 - \rho_1) g_{i-1} \tag{8}$$
$$t_i = \rho_2 t_{i-1} + (1 - \rho_2) g_{i-1} \odot g_{i-1} \tag{9}$$

where ρ_1 and ρ_2 are constants with typical values 0.9 and 0.999 respectively. The above estimates are corrected according to:

$$\widehat{s}_i = \frac{s_i}{1 - \rho_1^i}, \quad \widehat{t}_i = \frac{t_i}{1 - \rho_2^i}, \tag{10}$$

and the final update is performed as:

$$w_i = w_{i-1} - \mu \frac{\sqrt{\widehat{s}_i}}{\sqrt{\widehat{t}_i} + \epsilon} \tag{11}$$

where $\epsilon \approx 10^{-8}$ to avoid an ill-conditioned division. In a similar fashion, Adagrad uses only an estimation of the second-order moment, whereas in RMSProp , the first-order moment is estimated. In Adadelta, a dynamic learning rate estimator is furthermore used. We invite the reader to consult the aforementioned references for a detailed understanding of the algorithms.

III. Simulations

For the simulation setup, we have generated the following 2D dataset. We have generated a training dataset of 1000 samples distributed uniformly over $[0, 1]^2$, 100 samples for the validation dataset and 900 samples for the test set. Label 1 was attributed to samples within a circle centered in $(0.5, 0.5)$ of radius $1/\sqrt{2\pi}$.

The MLP architecture we considered has 3 layers of size $[16, 32, 2]$, i.e., the input layer has size $2 \to 16$, the hidden layer with size $16 \to 32$ and the output layer with size $32 \to 2$ in the Cross Entropy case and size $32 \to 1$ in the MSE case. In Fig. 1, we have first compared the performance over training epochs using the MSE loss (solid lines) and the cross-entropy loss (dashed lines). We have used throughout the simulations 100 epochs and batch size of 20. We used the uniform initialization strategy and the SGD optimizer for this first comparison.

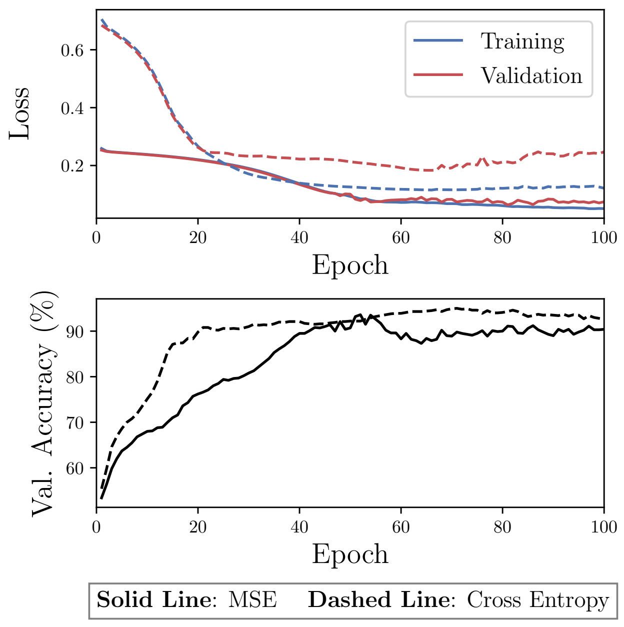

Figure 1. Average training and validation performance over epochs for the MSE and Cross Entropy losses using SGD updates.

Note that the values in Fig. 1 correspond to an average over 10 independent training instances. We notice that the cross-entropy minimization yields slightly improved validation accuracy, despite the worsened validation loss at final epochs.

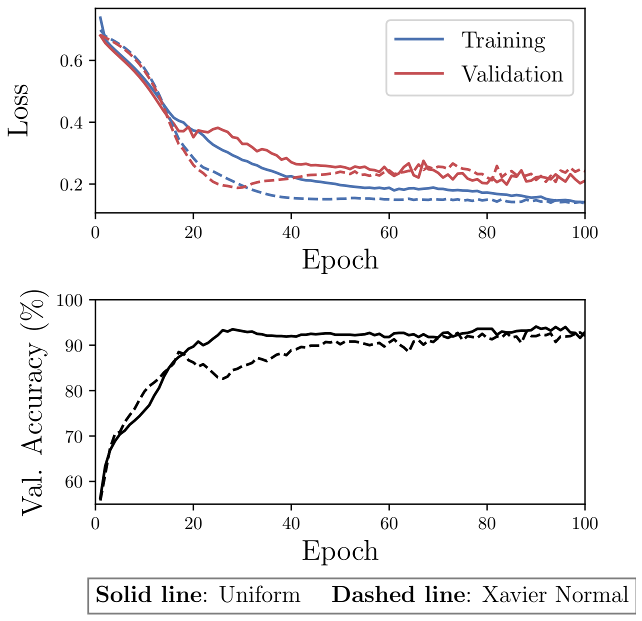

Figure 2. Average training and validation performance over epochs for the Xavier normal and for the uniform initialization strategies using the Cross Entropy loss with SGD updates.

Using now the cross entropy loss, and still considering the SGD optimizer, we consider the two proposed initialization strategies: Xavier normal and uniform initialization. In Fig. 2, we see the performance over training epochs for both initialization strategies. The same training conditions (batch size, number of epochs and architecture) were used for this experiment.

In Fig. 2, we see that between training epochs 20 and 40, we see an improved performance given by the Xavier approximation (see loss curves). This advantage vanishes over training epochs: towards the end of the training phase, there is no difference between the performance of both initializations.

Finally, we experiment with the different optimizers: SGD, Adam, Adagrad, RMSProp, Adadelta. For all methods we fix the learning rate as 0.001, and use the recommended values found in the literature for the remaining hyperparameters (Note that Adadelta does not require a learning rate specification). In Fig. 3 we see the (average) training performance for each of the optimizers. As expected, since SGD does not leverage dynamic adaptation according to the data, SGD is one of the slowest methods to converge (along with Adadelta). In terms of generalization, RMSProp and Adam seem to be the poorest performers, while Adagrad and Adadelta seem to achieve the best generalization scores. It should be noted that we did not optimize the choice of hyperparameters in each case, which is recommended in real applications.

For completeness, we also report the average test losses for each setup: SGD: 0.3411, Adam: 0.2466, Adagrad: 0.1008, RMSprop: 0.1589, Adadelta: 0.1282. The average test accuracies achieved are: SGD: 89.44%, Adam: 94.44%, Adagrad: 95.67%, RMSprop: 94.22%, Adadelta: 95.00%.

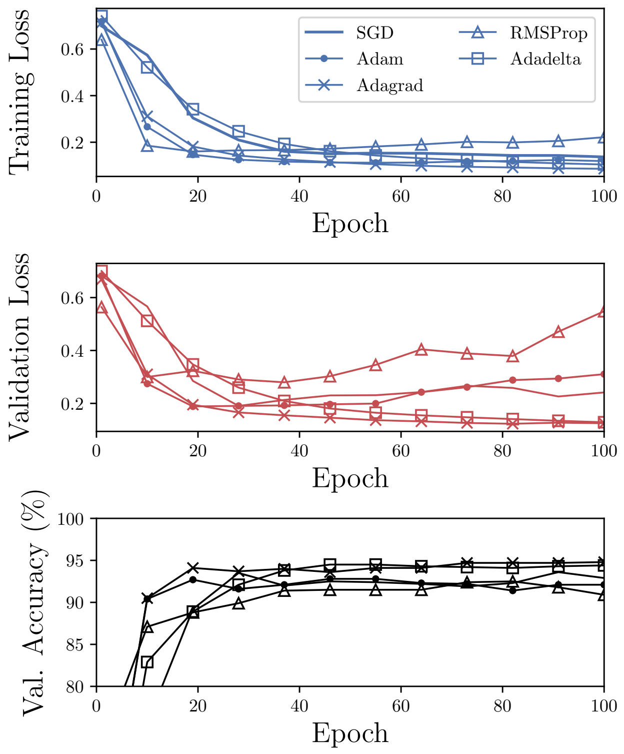

Figure 3. Average training and validation performance over epochs for the different optimizer strategies.

IV. Summary

In this work, we have implemented a mini deep learning framework, which allows for the training and evaluation of deep MLPs to minimize mean-squared error and cross-entropy losses. We have shown for a classification task, that the framework enables the correct use of different activation functions, initialization strategies and optimizers.

References

[1] F. Fleuret, "EPFL EE-559 Deep Learning Course," 2018, [Online]. Available: https://fleuret.org/dlc.

[2] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1, no. 2.

[3] M. D. Zeiler, "Adadelta: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

[4] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.