Project 2 - Mini deep-learning framework

Martin Everaert, Antony Doukhan

May 22, 2020

Contents

This report is 3 pages long without this cover page.

1 Introduction

The objective of this project was to develop a deep-learning framework only using basic `Pytorch`'s tensor operations and especially without the `Autograd` and neural-network packages. The framework was tested with a binary classification task on 2-dimensional vectors.

2 Framework

The structure of our framework is decomposed into modules that inherit from an abstract class `Module`. Any module overrides the `forward()`, `backward()` and `param()` methods. Parameters, such as weights and biases of linear layers are handled with a class `Parameters` where each parameter has a value and a gradient.

2.1 Linear modules

A fully-connected network has the following dynamics at each layer l:

$$\begin{cases} \mathbf{s}^l = \mathbf{x}^{l-1}\mathbf{W}^l + \mathbf{b}^l \\ \mathbf{x}^l = \phi^l(\mathbf{s}^l) \end{cases} \tag{1}$$

with $\mathbf{W}^l$, $\mathbf{b}^l$ denoting the weight and bias matrices of the `Linear` modules and ϕ^l an arbitrary activation function (`ReLu`, `tanH`).
It is possible to construct such a fully-connected network with the `Sequential`, `Linear` and activation-functions modules our framework provides. The test executable network is a `Sequential` module that is composed of `Linear` sub-modules as well as activation function sub-modules.

Initialization The parameters of a `Linear` module are the weights and biases tensors. The latter are defined as `Parameters` objects having a value and a gradient associated. Each weight tensor is of size $N_{l-1} \times N_l$. The weights and bias are initialized with standard gaussian distribution. As proposed in [1], we also provide an optional `Xavier` initialization of weights to avoid exploding and vanishing gradients: $W^l_{i,j} \sim \mathcal{N}\left(0, 2/(N_l + N_{l-1})\right)$.

Backpropagation According to our dynamic, the linear layer output backward pass is computed as :

$$\frac{\partial \ell}{\partial \mathbf{x}^{l-1}} = \mathbf{W}^l \frac{\partial \ell}{\partial \mathbf{s}^l} \quad ; \quad \frac{\partial \ell}{\partial \mathbf{W}^l} = \mathbf{x}^{l-1} \frac{\partial \ell}{\partial \mathbf{s}^l} \quad ; \quad \frac{\partial \ell}{\partial \mathbf{b}^l} = \frac{\partial \ell}{\partial \mathbf{s}^l} \tag{2}$$

where $\mathbf{x}^{l-1}, \mathbf{s}^l$ are respectively the input and output of the `Linear` module at layer l and ℓ denotes the loss function.

2.2 Activation functions

Implemented activation function modules are parameterless. However, it is possible to implement parametric activation functions (as any parametric module), such as `PReLU` in our structure.

2.2.1 Tanh

The `Tanh` module is simply applying the tanh function to every elements of the tensor. The backward pass consists in multiplying element-wise gradient with respect to output with the derivative given by $\tanh'(\mathbf{s}) = 1 - \tanh^2(\mathbf{s})$.

2.2.2 ReLU

`ReLU` module applies the rectifier function to every elements of the tensor. Its derivative is given by the Heaviside step function: $\frac{d}{dx}x^+ = \theta(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$ which is applied to each element of the input and multiplied element wise to the output gradient during the backward pass.

2.2.3 Softmax

For classification task, the output can be interpreted as a discrete probability distribution with the use of a `Softmax` activation function at last layer. The `Softmax` module converts the inputs to $[0, 1]$ outputs applying the function :

$$s_i \mapsto p_i(s_i) = \frac{1}{N}.e^{s_i - \max\limits_k\{s_k\}} \qquad ; \qquad N = \sum_k e^{s_k - \max\limits_k\{s_k\}} \tag{3}$$

The exponential control parameter $e^{-\max\limits_k\{s_k\}}$ is used to avoid overflows.
The gradients are computed as follows :

$$\frac{\partial p_i}{\partial s_k} = -p_i(\delta_{ik} - p_k) \tag{4}$$

2.3 Loss function

In addition to the required mean squared error (MSE) loss, we implemented a `CrossEntropyLoss` module since it is more suitable for classification task if coupled to a Softmax normalization. It is defined as:

$$\ell = -\sum_i t_i \log(p_i) \qquad ; \qquad \frac{\partial \ell}{\partial p_i} = -t_i/p_i \tag{5}$$

where t_i, p_i denote respectively the target and predicted probabilities.

2.4 Optimizer

We have implemented a `SGD` class to optimize parameters according to the stochastic gradient descent method. Based on [2] we also added a momentum method for gradient update.

3 Results of the test executable

Our test executable has runned successfully for 15 rounds of 25 epochs.
For each epoch, 1,000 points are sampled in $[0, 1]^2$ and are labeled 0 or 1 according to their distances to $(0.5, 0.5)$. The network is composed of 2 input units, 3 hidden `Linear` layers of 25 units with `TanH` activations and 2 output units.
We decided to compare the model with MSE loss vs CrossEntropy coupled with Softmax normalization and a fixed learning rate of 0.001. Provided estimations have been made through 15 rounds for each, using manual seed for data generation. For a fixed epoch, we compute the mean (*mean*) and the standard variation (*std*) of the 15 values. The plots show the mean value *mean* and the area from $mean - std$ to $mean + std$ is filled.

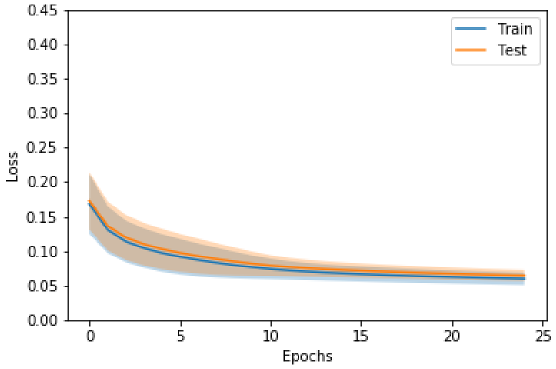

Figure 1: Evolution of loss over 25 epochs when trained with MSE

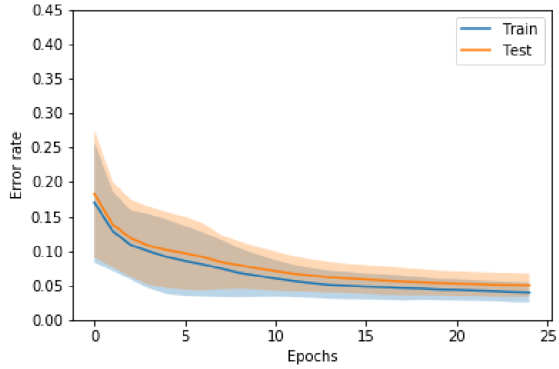

Figure 2: Error rate over 25 epochs when trained with MSE

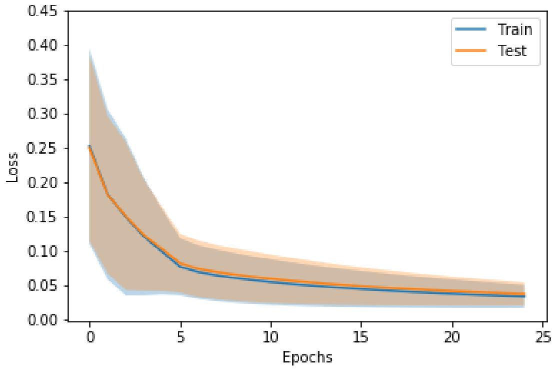

Figure 3: Evolution of loss over 25 epochs when trained with CrossEntropy and Softmax

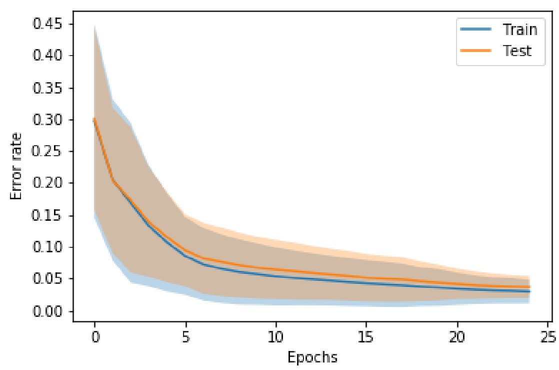

Figure 4: Error rate over 25 epochs when trained with CrossEntropy and Softmax

The average error after 25 epochs is overall relatively low for both experiment but slightly lower for CrossEntropy and Sofmax model ($\approx 3.7\%$ vs 5.0% for MSE). However the variance is much higher than using MSE loss during the learning.

4 Conclusion and possible improvements

Our mini deep-learning framework provides tools to train a fully connected network, optimizing parameters with SGD for MSE. In addition to the `CrossEntropyLoss` and `MSELoss` losses we have implemented, it would be possible to add other loss functions such as Hinge loss or L1-loss, as well as custom loss functions. Similarly, in addition to the modules `SoftMax`, `TanH`, `ReLU`, `Linear` and `Sequential` we have implemented, it would be possible to add other modules such as convolutionnal layers, dropout layers, as well as custom modules. It would also be possible to add other optimizers such as gradient descent or Adam. Note that for new optimizers and modules, it may be required to modify the existing modules to support batch computations.

References

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

[2] *On the importance of initialization and momentum in deep learning*, I. Sutskever, J. Martens, G. Dahl, G. Hinton