EE-559: Deep Learning - Project 2

Riccardo Cadei
riccardo.cadei@epfl.ch
SCIPER: 321957

Riccardo Fradiani
riccardo.fradiani@epfl.ch
SCIPER: 322979

Niccolò Polvani
niccolo.polvani@epfl.ch
SCIPER: 321215

***Abstract*—The aim of this project is to design and implement a mini Deep Learning framework based only on `PyTorch`'s tensor definition and elementary mathematical operations. Several modules are proposed (Fully Connected Linear Layer and Activation Functions) to create different Neural Networks (mainly Multi Layer Perceptron), define a Loss Function, compute the gradient with respect to it (using BackPropagation algorithm) and train the model using Batch Gradient Descent optimizer. Different experimental tests are conducted for a binary classification problem and it was solved with success ($TestError = 2.29\%$).**

I. Introduction

The aim of this project is to design and realize a mini "deep learning framework" using only `PyTorch`'s tensor definition and elementary mathematical operations. In particular, we want the whole framework to work without relying on `pytorch`'s autograd and neural-network modules.
Our final framework provides the necessary tools to:

- create a Multi Layers Percepton combining fully connected linear layers and activation functions (Tanh, ReLU and Sigmoid),
- Compute the gradient with respect to a loss function (MSE, MAE and Cross Entropy) using BackPropagation,
- optimize the model parameters using Stochastic Gradient Descent (SGD), also considering mini-batches.

The description of the general structure of the framework is described in Section II, the modules implemented are explained in Section III. In Section IV experimental tests are conduced to check the correctness of the implementation and final conclusions are reported in Section V.

II. Design

As a general structure for our framework, we decided to create an abstract `Module` class and then implement several modules and losses that inherit from it. In particular our `Module` class has the following methods:

- `forward`: gets for input, and returns, a tensor or a tuple of tensors, applying the module's transformation.
- `backward`: gets as input a tensor or a tuple of tensors containing the gradient of the loss with respect to the module's output, accumulates the gradient w.r.t. the parameters, and returns a tensor or a tuple of tensors containing the gradient of the loss w.r.t. the module's input.
- `param`: returns a list of pairs, each composed of a parameter tensor, and a gradient tensor of same size.

Some modules then present additional methods which serve specific functions. In particular, from this main class we implemented the modules: `Sequential`, `Linear`, `ReLU`, `Tanh`, `Sigmoid` and `Loss` (MSE, MAE and Cross Entropy), which will be described in the following section.
Moreover, we defined an abstract `Optimizer` class with only one method: `step`, which executes a Gradient Descent step, updating the model's weights. The class called `SGD` inherits from Optimizer and allow to update the model's parameter both a train example at time (SGD), or a batch at time (BGD or even GD).

III. Modules

A. Sequential

The module `Sequential` creates a neural network given a sequence of consecutive modules (layers). Its `Forward` and `Backward` attributes call iteratively the corresponding attributes for the single layers which compose the sequence, using as input respectively the output of the previous and following layer. By doing so, the back-propagation algorithm for the whole network is implemented. In addition to those and to the usual `Param` attribute, there is also an additional attribute `Zero_grad` which resets all the gradients of the network to zero when called.

B. Linear Layer

The `Linear` module defines a fully connected linear layer. It takes as mandatory arguments for its definition the input dimension (*input_dim*) and the output dimension (*output_dim*), while as optional arguments it is possible to state if a bias should be considered (by default *bias=True*) and how the weights of the layer should be initialized (by deafault *init_option='normal'*).

- *Parameters:* collects, in a list of couples, the parameters of the module (weight W and bias b) and the loss derivative with respect to them (dW, db).
- *Parameters initialization:* takes as input the initialization option and initializes the weights accordingly. The three options available for initializing the parameters are:
  - *Standard normal initialization ('standard')*: weights are initialized at random from a standard normal distribution
  - *Original Xavier initialization [1] ('xavier')*: weights are initialized at random from a normal distribution with $\sigma = \sqrt{\frac{2}{\text{input\_dim}+\text{output\_dim}}}$

  - *Resnet Xavier initialization [2] ('normal')*: weights are initialized at random from a normal distribution with $\sigma = \sqrt{\frac{2}{\text{output\_dim}}}$
- *Forward pass:* takes as input a tensor X and applies the linear transformation $s = WX + b$ where W is the weight tensor, b is the bias tensor and s the output tensor.
- *Backward pass:* takes as input the gradient w.r.t the output of the layer ($\frac{\partial L}{\partial s}$), then multiplies it by the input transposed (X^T), finding the derivative of the loss w.r.t the weights ($\frac{\partial L}{\partial W}$). The result is then added to the weights gradient parameter (dW). In particular:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial s} X^T \quad \text{and} \quad \frac{\partial L}{\partial b} = \frac{\partial L}{\partial s}$$

  Finally, it returns the gradient of the loss w.r.t. the input by applying:

$$\frac{\partial L}{\partial X} = W^T \frac{\partial L}{\partial s}$$

C. Activation functions

For the activation functions, each module is only characterized by a forward and a backward pass. Given an input X, elementwise we have:

ReLU:

- *Forward pass:*

$$\text{ReLU}(X_{ij}) = max(0, X_{ij})$$

- *Backward pass:*

$$\frac{\partial L}{\partial X_{ij}} = \frac{\partial L}{\partial s_{ij}} \cdot \text{dReLU}(X_{ij}) = \begin{cases} \frac{\partial L}{\partial s_{ij}}, & X_{ij} > 0 \\ 0, & \text{otherwise} \end{cases}$$

Tanh:

- *Forward pass:*

$$\text{Tanh}(X_{ij}) = \frac{e^{X_{ij}} - e^{-X_{ij}}}{e^{X_{ij}} + e^{-X_{ij}}}$$

- *Backward pass:*

$$\frac{\partial L}{\partial X_{ij}} = \frac{\partial L}{\partial s_{ij}} \cdot \text{dTanh}(X_{ij}) = \frac{\partial L}{\partial s_{ij}}(1 - \text{Tanh}(X_{ij})^2)$$

Sigmoid (σ):

- *Forward pass:*

$$\sigma(X_{ij}) = \frac{1}{1 + e^{-X_{ij}}}$$

- *Backward pass:*

$$\frac{\partial L}{\partial X_{ij}} = \frac{\partial L}{\partial s_{ij}} \cdot \text{d}\sigma(X_{ij}) = \frac{\partial L}{\partial s_{ij}}(\sigma(X_{ij}) - \sigma(X_{ij})^2)$$

D. Loss functions

The class called `Loss` inherits from `Module` and therefore implements the 2 methods `forward` and `backward`.

- *Forward pass:* compute the Loss using MSE, MAE or CrossEntropy. It takes as inputs the model's predictions $\hat{Y}$ and the ground truth Y.
- *Backward pass:* calculates the gradient of the loss function with respect to the model's outputs $\frac{\partial L}{\partial Y_i}$ and calls `model.backward`, passing this value as input.

In particular, let n the number of examples:

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^{n} |Y_i - \hat{Y}_i|$$

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2$$

Let m the number of classes to predict and expressing the output and the ground truth in one-hot-encoding form:

$$\text{L}_{\text{CE}} = -\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{m} Y_{ij} \log(p_{ij})$$

where

$$p_{ij} = \frac{\hat{Y_{ij}}}{\sum_{k=1}^{m} \exp{(\hat{Y_{ik}})}} \qquad \forall i, j$$

E. Optimizer

SGD inherits from Optimizer and defines `step` as:

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \gamma \nabla L(\mathbf{w}_n)$$

where $\mathbf{w}$ represents the tensor of model's weights, γ is the learning rate, and $\nabla L(\mathbf{w})$ is the gradient of the loss with respect to the weights. Depending if only an example or several examples are passed in input, then Stochastic Gradient Descent or Batch Gradient Descent is performed (in particular considering $batch\_size = n$ we have Gradient Descent)

IV. Experimental tests

The framework is then tested on a binary classification task using a Multi Layer Perceptron.

The dataset used consists in a training and a test set of 1000 points each, sampled uniformly in $[0, 1]^2$, each with a label 1 (one-hot-encoded) if inside a disk centered at $(0.5, 0.5)$ of radius $\frac{1}{\sqrt{2\pi}}$, and 0 otherwise. We then split the training set in two parts: 90% of it for training to update the model's weights, while the remaining 10% was kept for validation. We used the validation set for the hyper-parameter tuning of the number of the epochs, batch size and learning rate.

We created a Multi Layers Perceptron with 3 hidden fully connected linear layers of 25 neurons each, all of them followed by the $ReLU$ activation function, and a final fully connected linear layer going to the $output\_size = 2$ followed by a sigmoid.

We trained the model weights on the training set for 300 epochs using Batch Gradient Descent with batch size equal

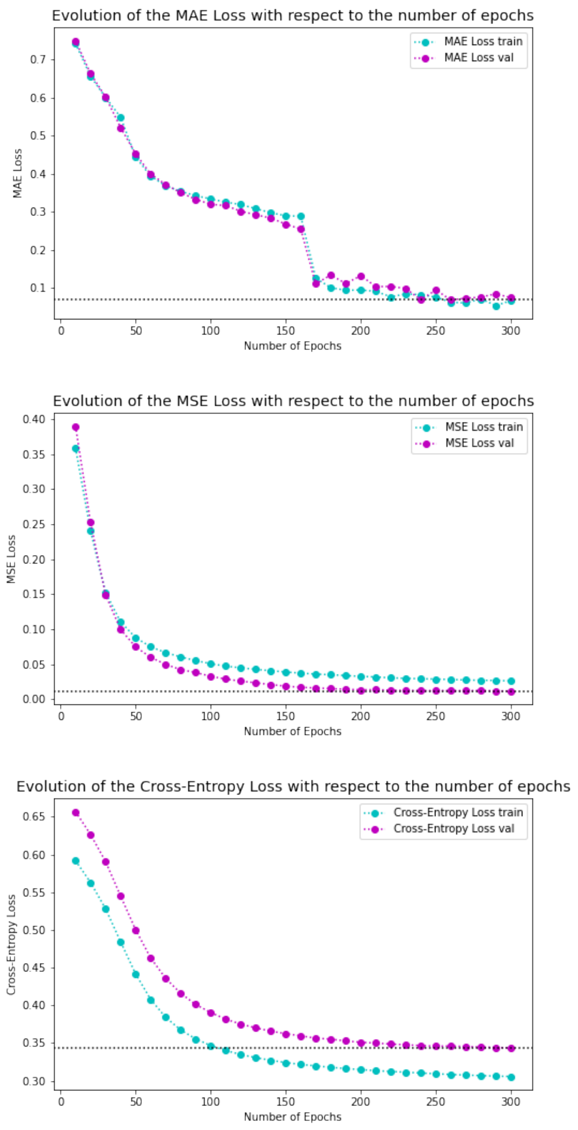

Figure 1: Evolution of the loss during training, using Mean Average Error (above), Mean Square Error (middle) and Cross-Entropy loss (below).

to 50 and learning rate equal to 0.001 comparing the performances using the three different loss functions. To perform Batch Gradient Descent: first we set to 0 the accumulated gradient of the model's weights, then we computed the loss on the predictions for the whole batch (forward pass) and executed the backward pass in order to compute the gradients, finally we executed a gradient descent step using the optimizer.

The evolution of the loss during the training and validation procedures for the 3 loss functions previously described are represented in Figure 1.

The performances of the same three experiments executed 10 times each are reported in Table I. In all the experiments the test set error in significantly smaller than 5% and in particular the best performances are obtained with $CrossEntropyLoss$, which provides mean error rate equal to 2.29%. For the

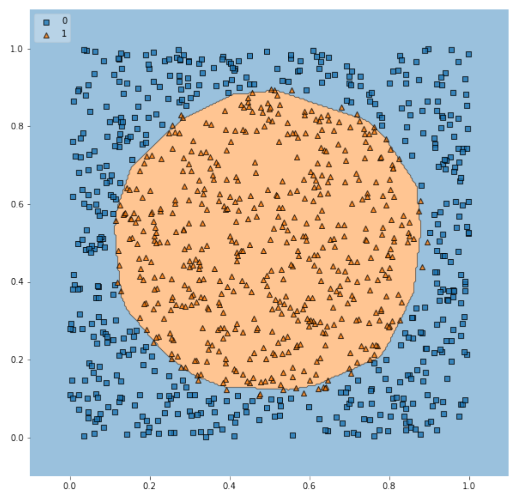

Figure 2: Decision region of the Multi Layer Perceptron trained using Cross-Entropy loss

.

experiment using the $CrossEntropyLoss$ we also report in Figure 2 the decision region of the trained model, which seem to reproduce very accurately the true decision region defined mathematically.

Loss	Train Error	Test Error
MSE	0.0162 ± 0.0051	0.0255 ± 0.0051
MAE	0.0232 ± 0.0085	0.0312 ± 0.0094
Cross Entropy	0.0162 ± 0.0068	0.0229 ± 0.0082

Table I: Results on train and test datasets for different loss functions

V. Conclusions

We designed and implemented a mini deep learning framework based only on `PyTorch`'s tensor definition and elementary mathematical operations. Several abstract and easy to scale classes are proposed. As future developments, new neural networks modules (e.g. Convolutional Layer, Batch Normalazation, Drop Out) and machine learning methods (e.g. Adam optimizer) could be added. The correctness of the implementation is confirmed by several experimental tests on a a 2-dimension non linear binary classification task.

References

[1] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, Y. W. Teh and M. Titterington, Eds., vol. 9. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. [Online]. Available: http://proceedings.mlr.press/v9/glorot10a.html

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385