Implementation of a simple deep learning framework

COLBOIS Laurent, HERNANDEZ Sergio Daniel, REISE Wojciech
Deep Learning Course, EPF Lausanne, Switzerland

***Abstract*—The framework presented in this report is the result of an exercise. The implementation is based on the Tensor data-structure present in PyTorch. The PyTorch framework is used as a model and inspiration. Our implementation features the modules allowing to perform a simple classification task, on which our framework is tested for accuracy. The speed and memory use are compared with the PyTorch reference implementation. Finally, possible flaws are listed along with how to overcome them and extend the framework**

I. INTRODUCTION

The aim of this project was to design a rudimentary Neural Network framework, that could perform well on a subset of those tasks. More precisely, we provide a framework that deals with binary classification.

We will start by introducing the standard procedure for the problems outlined above and specify what it looks like in the case of classification. This will provide motivation for the architecture of our modules. Later, we provide mathematical definitions of the necessary modules and how they are implemented. What follows next is an analysis of the performance of our framework and finally, the possible improvements.

II. STANDARD MACHINE LEARNING PROCEDURE

In general, supervised machine learning tasks consist of approximating a function $f^* : \mathbb{R}^d \to \mathbb{R}^n$, that we know only on a small subset of the domain. To approximate it, we first choose a family of functions $\mathcal{F}$. Then, we need a systematic way of finding a good approximation $\hat{f} \in \mathcal{F}$ of f^*.

More formally, all information on f^* is captured as a set of pairs $\{(x_i, y_i)\}_{i=1}^N$, where $f^*(x_i) = y_i$. Note that the existence of an underlying f^* is assumed implicitly and it is only true if x_i contains enough information to determiny y_i. Usually, the family $\mathcal{F}$ is chosen to be a composition of simple linear functions and activation functions. Each such function will be called a layer and the composition of them is a called a neural net (NN). The richness of $\mathcal{F}$ is determined by the amount of parameters in the neural net. In current architectures, dim($\mathcal{F}$) may be of the order of 10^6 and has the structure of a vector space. Hence, an automated procedure is needed to find a good representative in $\mathcal{F}$. To do this, the available data is generally divided in two parts, called the train and the test set respectively. The former is used to select the function $\hat{f}$, while the latter serves solely to evaluate how general $\hat{f}$ is, and should never be used for the selection.

The determination of a good function is an optimization problem that is treated using the *backpropagation* algorithm. This idea is to compute the output $\hat{y}_i$ produced by the NN for an input x_i (forward pass), use a loss function to evaluate how far $\hat{y}_i$ is from y_i, and to update the parameters of the NN in order to decrease this loss. A convenient update is generally computed using variants of the Gradient Descend algorithm. The computation of gradients is called the *backward pass*.

The above concepts outline a structure for the framework. First, we need a class `MLP`, that implements a neural network. This class will contain several `Modules` that represent the composition of functions we have chosen. Instances of an `OptimAlgo` will then take care of iteratively choosing better and better f, such that we get $\tilde{f}$, "close" to a best representative $\hat{f}$. Finally, we evaluate how well $\tilde{f}$ performs using an instance of a subclass of `Loss`.

For the optimization of the NN, since the family can be represented as a vector space of parameters, we can use standard optimization techniques based on the gradient, that work in the continuous setting. Hence, we need a mechanism to calculate the gradients of the loss function, with respect to the parameters. Recall that, after all, a NN is a composition of functions, so computing those gradients is relatively easy, as they are just one can simply repetitively apply the chain rule for derivation : during the backward pass, we simply need to calculate and propagate the gradient from the last layer to the earliest one (the one which takes the input of the NN as argument). We have implemented and included explanations for this procedure in this report.

III. MODULES

The components of the simplest neural networks are linear layers, separated by non-linear activation layers. We implemented the linear layers and the following activation functions: ReLU, LeakyReLU and Tanh. We are first going to describe the mathematics behind each of the modules.

A. Linear

Recall that our input $x_i \in \mathbb{R}^d$. Assume that we have N samples. Hence, $(x_i)_j$ can be seen as a matrix $x \in \mathbb{R}^{d \times N}$. A linear function can then be expressed as $y = Wx + b$, where $W \in \mathbb{R}^{n \times d}$, $b \in \mathbb{R}^n$. The parameters that remain to be fixed in such a module are the coefficients of W, b. Let us denote by $\mathcal{L}$ the loss obtained on x through the current function f and then give the expressions for the gradients with respect to the input and the parameters.

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y}\,\frac{\partial y}{\partial x} = \frac{\partial \mathcal{L}}{\partial y}\,W \tag{1}$$

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial y}\,\frac{\partial y}{\partial W} = \frac{\partial \mathcal{L}}{\partial y}\,X^T \tag{2}$$

$$\frac{\partial \mathcal{L}}{\partial b} = \sum_{i=1}^{N}\left(\frac{\partial \mathcal{L}}{\partial y}\right)_{:,i}, \tag{3}$$

where $\frac{\partial \mathcal{L}}{\partial y}$ is seen as a matrix $\mathbb{R}^{n\times N}$ and is the gradient propagated up to the current layer during the backward pass.

B. LeakyReLU and ReLU

The LeakyReLU (Leaky Rectified Linear Unit) is a (non-linear) activation function:

$$x \mapsto \begin{cases} x & x \geq 0 \\ \alpha x & x < 0 \end{cases} \tag{4}$$

where $\alpha > 0$ is a fixed parameter. Like all the activation functions, it has no parameter to be optimized over. However, we still need the gradient of the loss with respect to the input, since it has to propagate "through" this layer:

$$\left(\frac{\partial \mathcal{L}}{\partial x}\right)_{i,j} = \begin{cases} \left(\frac{\partial \mathcal{L}}{\partial y}\right)_{i,j} & x \geq 0 \\ \alpha\left(\frac{\partial \mathcal{L}}{\partial y}\right)_{i,j} & x < 0. \end{cases} \tag{5}$$

This can be written more generally as

$$\frac{\partial \mathcal{L}}{\partial x} = (\mathbb{1}_{x\geq 0} + \alpha \mathbb{1}_{x<0}) \odot \left(\frac{\partial \mathcal{L}}{\partial y}\right)_{i,j} \tag{6}$$

, where $\mathbb{1}_{x\geq 0}$ is a matrix with coefficient $(i,j) \neq 0 \leftrightarrow x_{i,j} > 0$ and $\odot$ denotes element wise multiplication.

The more popular ReLU (Rectified Linear Unit) is a particular version of the LeakyReLU, where $\alpha = 0$. Hence, the gradient keeps only those components, where the input was positive.

C. Tanh

Another activation function implemented in our framework is the hyperbolic tangent:

$$x \mapsto \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

This function, applied to a tensor, is taken component-wise. For the forward pass, notice that if $|x|$ is big, then we may be in an overflow situation. A way to deal with that is to rewrite the function as $\frac{1-e^{-2x}}{1+e^{-2x}}$ and $\frac{e^{2x}-1}{e^{2x}+1}$ for $x > 0$ and $x < 0$ respectively. However, in the case of this project, we did not encounter this issue, hence, it is not implemented so as not to add complexity. Now, similarly to the (Leaky)ReLU, there are no parameters to be optimized over here, so in the backward pass, we only need to propagate the gradient. The derivative is given by $x \mapsto 1 - \tanh(x)^2$. On the whole dataset, the backward pass through a hyperbolic tangent writes

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial x}\left(1 - \tanh(x)^2\right) \tag{7}$$

where $1 \in \mathbb{R}^{size(x)}$is a tensor of the same size as x and filled with ones.

IV. Loss functions

As mentioned in the introduction, we need to evaluate how well the model performs in order to correct it. For this, we give 3 possibilities: MSE, Cross Entropy loss and Classification loss.

A. MSE

MSE stands for Mean Squared Error and is one of the commonly used cost functions $\mathcal{L} = \mathcal{L}_{\text{MSE}}$ for optimization problems in general. For a model m, in our case a NN, we have

$$\mathcal{L} : (x_i, y_i)_{i=1}^N \mapsto \frac{1}{N}\sum_{i=1}^{N}(m(x_i) - y_i)^2 \tag{8}$$

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{2}{N}\sum_{i=1}^{N}(m(x_i) - y_i) \tag{9}$$

B. Cross Entropy

A loss function that is commonly used in classification problems is the Cross Entropy. Given the nature of our task, our framework supports only binary classification Cross Entropy, on labels $\{0,1\}$ and is hence defined:

$$\mathcal{L} : (x_i, y_i)_{i=1}^N \mapsto \frac{1}{N}\sum_{i=1}^{N} -(y_i \log(\tilde{x_i}) + (1-y_i)\log(1-\tilde{x_i})) \tag{10}$$

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{2}{N}\sum_{i=1}^{N} y_i e^{-m(x)}\tilde{x_i} - (1-y_i)e^{-m(x_i)}\frac{\tilde{x_i}^2}{1-\tilde{x_i}}, \tag{11}$$

$$\text{where } \tilde{x_i} = \frac{1}{1+e^{-m(x_i)}} \tag{12}$$

C. Classification

Finally, we have implemented a loss module which counts the number of misclassified samples. Implementing this loss as a subclass of `Loss` is, conceptually, the best solution to integrate this functionality in our framework. The drawback of this solution is that it does not implement a `backward` method - `ClassificationLoss().backward(model)` yields a `InvalidOperation`. For a binary classification problem, this classification loss receive a two-dimensional output from a model, and takes the index of the most active dimension as the class label. If this label does not match with the corresponding training label, the loss is incremented by one.

V. Optimization algorithms

Finally, once we have made a forward pass and evaluated the loss associated to the output, we might want to use this information to improve our model. In the framework, two standard optimization algorithms are proposed.

A. Gradient Descent

The gradient descent algorithm apply one forward and one backward pass on the whole train dataset (which is called one epoch) to compute the gradient of the loss with respect to each parameter of the model, and then compute for every parameter w the update

$$w = w - \gamma \frac{\partial \mathcal{L}}{\partial w} \tag{13}$$

where γ is called the learning rate and should not be to large. This optimization algorithm is not very good for two main reasons : an optimization step is done only once every epoch and so convergence is very slow, and the fact that the gradient is computed with respect to all samples increases the risk to get trapped in a local minimum, which would result in a model with bad generalization ability. Those problem are solved by the use of the *Stochastic Gradient Descent* (SGD).

B. Stochastic Gradient Descent

The SGD runs a the backpropagation algorithm only on a small batch of samples, randomly selected, before performing an optimization step. This imply convergence is faster (more steps per epochs), and also that the noisier gradient leaves some chance of getting out of local minimas during the training. The random selection of samples is done by simply shuffling the sample before each epochs, and then feeding the data to the NN by mini-batches. This mean each sample is used exactly once per epoch. We also implemented as a curiosity the `SGDWithRepetition`, which systematically replaces the samples as candidate for the batch selection. This mean that the notion of epochs becomes less clear, as some samples can be used several times while another has not been yet.

VI. Implementation

We describe our architecture in a top to bottom manner - starting from the objects we want to manipulate directly. In the framework, one creates a NN model by calling either the `MLP()` or `Sequential()` constructors with appropriate parameters - the latter is just a subclass, providing an alternative constructor. It could have been implemented as another method in `MLP`, but we decided to offer the possibility of extending `Sequential` if needed. A `MLP` is just a list of `modules`. Any `module` has to be able to

1) perform a forward pass,
2) perform a backward pass, given a gradient coming from a loss function, or another module,
3) update it's weights.

The `MLP` simply delegates those tasks to its `module` list.

The loss functions mentioned calculate the discrepancy between the output and the target and, more importantly, provides the initial gradient for backpropagation. Note that this is the case only for `MSE` and `CrossEntropy`, while `ClassificationLoss` serves only the first purpose, as it is not differentiable.

All in all, the training procedure is implemented by the optimizer in `OptimizeAlgo.train_model`. We notice that it only loops on epochs and delegates to subclasses the task of performing one epoch. The reason for this is that this loop is redundant for all optimizers. The only thing that changes is what happens in a particular epoch - for example, taking mini-batches or shuffling the input samples.

VII. Performance

We have evaluated the accuracy of our framework on a toy dataset and compared it's performance to the reference framework - PyTorch. Our comparison is 2-fold: speed and memory.

The dataset on which the tests are performed is comprised of 1000 data-points, sampled randomly, uniformly and independently in $[0,1]^2$. The data-points are labeled 1 if they lie inside the ball of radius $\frac{1}{\sqrt{2\pi}}$ centered in (1/2,1/2), and 0 otherwise. The distribution of labels is a Bernoulli with probability $\frac{1}{2}$.

A. Accuracy

The model we have used to evaluate the performance of our framework is the one suggested in the project description. Namely, a neural network with two hidden layers of input and output sizes of 25. The input layer is then a 2 to 25 layer while the output is a 25 to 2 layer. All these fully-connected linear layers are separated by activation functions - rectified linear units. To be more precise, the code reads as follows:

```
m.Sequential(m.Linear(2,25), m.ReLU(),
             m.Linear(25,25), m.ReLU(),
             m.Linear(25,25), m.ReLU(),
             m.Linear(25,2))
```

B. Speed

First, notice that objects (models, loss functions and optimizers) in our framework are quicker to initialize, compared to PyTorch. On the other hand, our forward and backward passes are significantly slower (see figure 2). For forward passes, this may be related to memory allocation: for each module, we save the input, in order to be able to evaluate at the right point during the backward pass. Also,

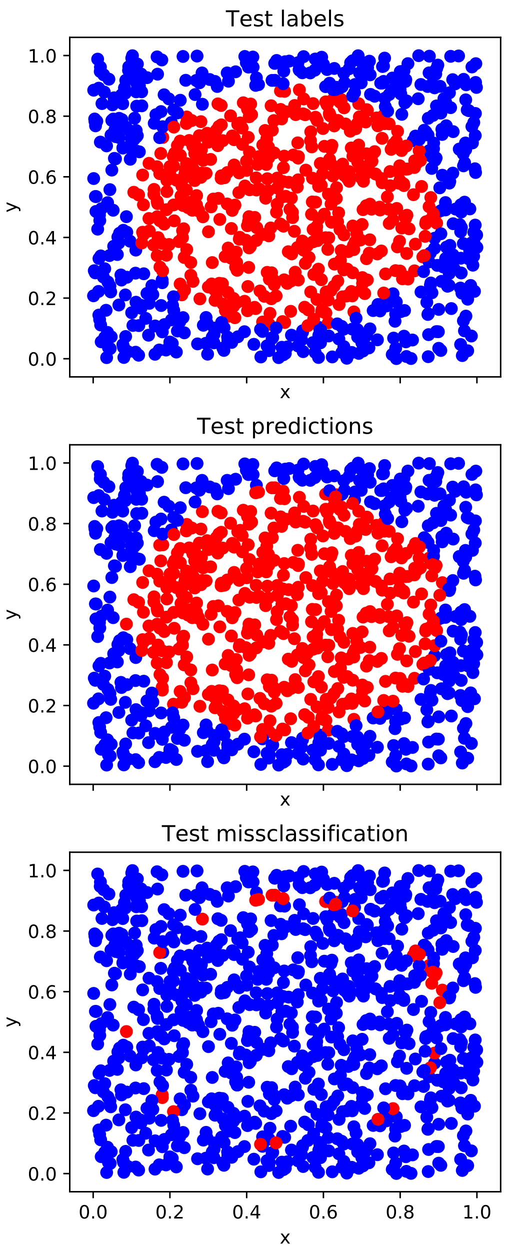

Fig. 1: *Classification performance on a randomly generated disk dataset. On the first and second plot, the colors correspond to the two classes. On the last plot a red color indicates a point of the test set that was missclassified. The total missclassification error rate is of 3.5%.*

PyTorch compromises initialization time to achieve better performance on passes for successive epochs.

	Our framework	Pytorch
Initialization	$0.90{\cdot}10^{-3}$	$4.72{\cdot}10^{-3}$
Forward Pass	$1.11{\cdot}10^{-6}$	$0.47{\cdot}10^{-6}$
Backward Pass	$1.37{\cdot}10^{-6}$	$0.64{\cdot}10^{-6}$

TABLE I: *Comparison of the execution times (in seconds for Initialization, [seconds/1000 samples] for the latter categories) between our framework and PyTorch.*

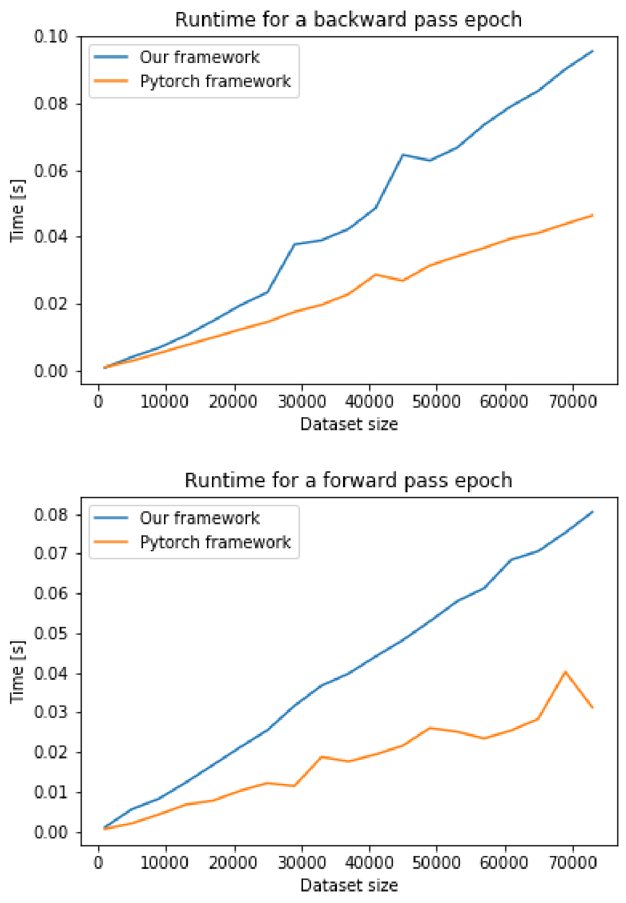

Fig. 2: *Comparison of the speed of our framework and PyTorch, on two time-consuming procedures in deep learning. Experiments performed on the toy data-set described above. For each sample size, the timing was performed 20 times to obtain a reliable average.*

C. Memory

Another interesting measure of performance is provided by memory consumption. One can look at how much memory is allocated to the process at any given time. Unfortunately, these measurements turn out to be surprisingly unreliable. Indeed, in the initial few epochs, the models tend to require more and more memory until it converges. Even then though, the memory usage varies greatly with execution, so no conclusion is possible. Nevertheless, it is clear that our model uses the same magnitude of memory, for the same dataset and model architectures.

VIII. Improvements and extensions

Many of the choices we have made in the design of the network impose some limitations on the use and possible extensions. In this section, we briefly present the drawbacks and how to circumvent them, and we briefly mention the possible improvements that our framework could benefit from.

First of all, during the backward pass, each model stores the gradients with respect to it's parameters. This is done, at each call of `optim.backward(model)` by replacing the old gradient value by the new one. This approach has two flaws. First of all, to be sure that the model is optimized for the set of samples that went through the forward pass, we need to call `loss.backward(model)` followed by `model.update_weights(optim)`, right after the forward pass. Then, it also makes it impossible to represent "weights linking". Overall, our implementation suffices in the case of the required modules and the present dataset, but could be improved by summing the successively calculated gradients and setting them to 0 after the optimization.
Another issue is how information is stored for backpropagation computation. In each module, we have a copy of the input data, made during the forward pass. In general, it is not a problem, but it could be one, in case of a very large network or input data. Indeed, first, if we have a big input to train over, it is easiest to process it in batches. That way, only the mini batch is copied and optimized over. This is generally good practice also from the point of view of model accuracy. The scenario in which the model is complex and having that many copies of the data is not possible, one could also imagine a batching strategy. During the forward pass, store the input every (batch size)-layers. Then, starting from the last "checkpoint", compute the forward pass (again) until the next checkpoint, saving all the inputs this time (at most (batch size) of them). Now, backpropagate the gradient as long as you can, so until the previous "checkpoint", and delete the previously stored inputs. Go one "checkpoint" back and compute the forward to the place where the gradient currently is. Continuing like this, we do exactly 2 forward passes for one backward pass.