Project 2: Mini deep-learning framework

Jonathan Labhard, Robin Zbinden, Jalel Zghonda
EPFL
EE559 - Deep Learning

***Abstract*— This is the report for Project 2 from EE559 Deep Learning class. In this project we design a mini deep learning framework from scratch, using only Pytorchs tensor operations and the standard math library from python. This framework enables us to build simple Neural Networks with an arbitrarily number of fully connected layer and to train it using Stochastic Gradient Descent. We also show that it can be used to solve almost perfectly a basic binary classification problem, and can be naturally extended to other more complex problems. We describe the different components and the results of this framework in this report.**

I. Introduction

Deep Learning frameworks such as Pytorch are now highly used to solve various sort of problems. These frameworks enable us to use complex Neural Networks in a simple way, asking us only to define the structure of the network wanted.

The goal of this project is to produce a similar framework to build Neural Networks composed of fully connected layers and distinct activation functions, and use Stochastic Gradient Descent to train them to solve different machine learning problems. The strength of this framework resides in its simplicity, as is can be seen in the section II.

II. The framework

A. Modules

The framework has globally the same interface as Pytorch. There are a few modules that you can combine to form a Neural Network:

- `Linear`: represents a linear fully connected layer, where the input and the output dimensions need to be precised.
- `ReLU`, `Tanh`, `Sigmoid`: are activation functions to introduce non-linearity after a `Linear` module for instance.
- `Sequential`: is used to combine different modules. The dimensions of the inner modules naturally need to match.

Here is a basic example of how to use them:

```
l1 = Linear(2,25)
a1 = ReLU()
l2 = Linear(25, 25)
a2 = Tanh()
l3 = Linear(25, 25)
a3 = Sigmoid()
l4 = Linear(25, 2)
model = Sequential(l1, a1, l2, a2,
                   l3, a3, l4)
```

These modules all extend from the `Module` abstract class and implement the following functions:

- `forward`: corresponds to the forward computation of the module for a given input.
- `backward`: corresponds to the backward pass, computing the gradient of the loss with respect to the input of the module and its parameters. The input of the module is obtained via the forward computation and the `backward` function must therefore be called after the `forward` function.
- `param`: returns a list of the parameters of the module (returns `None` for the activation functions) and their corresponding gradients.
- `update_param`: updates the parameters of the module (if it has parameters) according to its gradients and a given learning rate.

In particular, the `Sequential` module's function sequentially calls the respective functions of each of its modules.

B. Training

After defining the structure of the Neural Network using the modules, the method `train_model_SGD` can be called to train the

model on a given training set using Stochastic Gradient Descent. Different parameters can be given such as the number of epoch, the size of the mini-batchs, the learning rate of the SGD and the type of the loss, which can be either Mean Squared Error (MSE) or Mean Absolute Error (MAE). It is also possible to use these two losses outside the `train_model_SGD` function to compute the loss or the gradient of the loss with respect to a given output of a model and a target set. Finally, the method returns a list of losses, computed for each epoch.

One can easily verify that its model generalizes well on new data by calling the method `accuracy` on a test set, which computes the accuracy of the predictions of the model on a given input set and target set.

Finally, the reader is also encouraged to go through the docstrings of the different functions to know more about them.

III. Testing Model

We test our framework with a basic sequential architecture with three hidden layers of size 25. Furthermore, we use the three different activation functions implemented to make sure they are functioning as intended. The specifics of the architecture are shown in table I.

We train the model using MSE Loss with 300 epochs on a simple toy dataset which is 1000 datapoints sampled uniformly in $[0, 1]^2$, each with a label 0 if outside the disk centered at $(0.5, 0.5)$ of radius $1/\sqrt{2\pi}$, and 1 inside. We then test on 1000 others datapoints sampled in the same way and shown (after normalization) on figure 1.

Layer operation	Parameters
Input	(1x2) Vector
Linear Fully Connected	(2, 25)
Activation function	ReLU
Linear Fully Connected	(25, 25)
Activation function	Tanh
Linear Fully Connected	(25, 25)
Activation function	Sigmoid
Linear Fully Connected	(25, 2)
Output	(1x2) Vector

TABLE I

Architecture used for testing the framework.

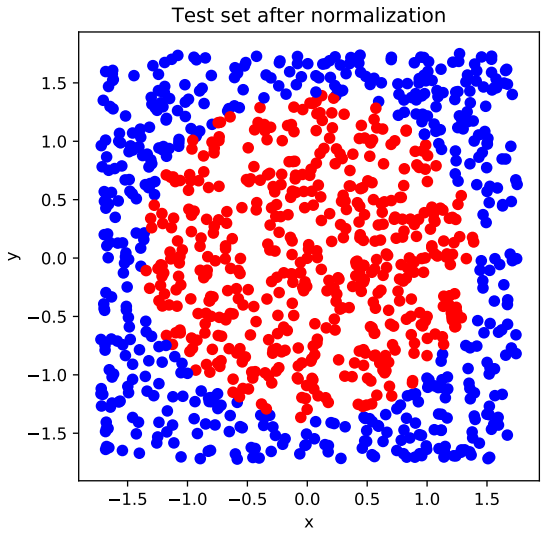

Fig. 1. Toy dataset after normalization used for testing the framework.

IV. Results

We choose the accuracy metric to attest the quality of the models, since we deal with a binary classification problem and that we have approximately the same number of datapoints in each binary class. The following performance estimates are based on the mean and standard deviation of the test accuracy obtained from the 10 different rounds of the training phase.

As we can observe from table II, we obtain very convincing results for both training and testing with a negligible variance. From these we can safely assume our framework is working properly and as intended on basic machine learning problems.

	Mean	σ
Train Accuracy	0.993	0.003
Test Accuracy	0.980	0.006

TABLE II

Results obtained in 10 rounds of training.

V. Conclusion

Through this report we build a framework that is relatively basic, but nonetheless efficient and modular. This allows the user to build on top of it and adapt it to a variety of different problems.

We design the different parts of the framework such that they are intuitive and easy to use. We

believe the framework presents a solid basis to what a neural network framework should be able to provide.

Finally, we run a model multiple times to make sure it is behaving as intended and reach satisfying and stable results.

To go further in the building of a framework we could improve it such that it can be used for more complex problems. We could for example implement more modules allowing to use convolution layers or other widely used operations. Other losses and optimizers could be considered as well since MSE, MAE and SGD are not always optimal, specifically for a classification task. Such improvements can be useful to further understand the ins and outs of a neural network framework, and can safely be built on top of the framework presented here.