EE-559 Deep Learning course : Project 2 report

Mini deep-learning framework

Samuel Dubuis (259157)
IC - EPFL
s.dubuis@epfl.ch

Yann Gabbud (260036)
IC - EPFL
yann.gabbud@epfl.ch

Vincent Gürtler (263880)
IC - EPFL
vincent.gurtler@epfl.ch

I. Introduction

The objective of this project was to recreate a miniature deep learning framework, using only the basic tensor object of Pytorch and the standard math library.

We decided to follow the suggestion given in the project explanations, which is to define a class named Module, based on a simple structure :

```
class Module(object):
    def forward(self, *input):
        raise NotImplementedError
    def backward(self, *gradwrtoutput):
        raise NotImplementedError
    def param(self):
        return []
```

Then, each of the different useful modules will inherit from it.

A. Structure of the framework

Our main idea was to recreate a neural network structure that distributes the computations into the main module `Sequential` which calls every other module, in sequence.

These lower tier modules are first the `Linear` one that mainly creates the layer of our neural network, then we have the activation functions `Sigmoid`, `ReLu`, `LeakyReLu` and `TanH`.

The optimizer used during the training will be using stochastic gradient descent `SGD`, also defined as a class but not inheriting the main module. It will store all the parameters of the model to perform the gradient descent step at each training batch.

Finally, we wrote a last module for the loss, `LossMSE`, for the mean-squared error loss and its derivative.

B. Methods

For the implementation, we focused on the training and testing part of our neural network.

The `train` method was created as such :

```
def train_model(model,optimizer,X_train,
    y_train,X_val,y_val,epochs,batch_size)
```

It took simple parameters as input, which are all self-explanatory. It was based on two parts: the actual training and the validation, used to avoid overfitting.

For each epoch and training's mini-batch, the model forwards the batch through the modules as explained previously, computes losses, backwards the derivative of the loss in all modules, and finally, the optimizer performs the gradient descent step.

Also, for each same epoch and validation's mini-batch, we compute validation losses based on the same principles, and the SGD optimizer modifies the learning rate if needed.

The `test` method was defined as such :

```
def test_model(model,X_test,y_test,
    batch_size)
```

This is the simplest possible testing method: for each mini-batch comparing the output with its ground-truth value and computing the total loss and accuracy.

Finally, a few other methods were created that we will not explain in detail as their name should be self-explanatory.

```
def generate_data(n):
def split_train_test(X,y,train_size=0.8,
    shuffle=True):
def normalize(data):
def one_hot_encode(y):
```

II. Modules

In this section we will present the different modules used in our framework.

A. Sequential

The module `Sequential` represents the neural network implemented in this mini-project, which is only composed of fully connected layers. Its main function is to handle and redistribute the computation into the modules its composed of, all in sequence. It takes as input the list of other modules the network will be made of.

The `Sequential` module also implements the inherited `param` method which returns all the parameters for each module in its sequence. These parameters can then be fed

to the optimizer which will take care of the gradient descent. This will be further explained in the section III.

B. Linear

The module `Linear` takes two integers as input. These integers represent the input and output dimensions of the fully connected layer create; where the weight vectors and the biases are implemented. They are all initialized by default with a standard normal distribution.

The `forward` method calculates the output by multiplying the weights and the input and then adding the bias. The `backward` method accumulates the gradient of the parameters such that the optimizer will be able to perform the gradient descent step.

C. Activation Functions

We implemented four different activation functions so that we could test and display the best result later on. An activation function is added in the `Sequential` module, always after the `Linear` in order to give some non-linearity to the model. Without the activation functions, the neural network could only perform linear mappings and could never learn complex functions.

1) ReLu: $max(0, x)$
2) LeakyReLu: $max(\alpha * x, x)$
3) TanH: here we called the function pre-implemented in PyTorch
4) Sigmoid: $\frac{1}{1-e^x}$

D. Loss Functions

In this section, we will explain briefly the loss function we used all throughout this mini-project. We implemented the mean-squared error loss. The formula is :

$$(\frac{1}{n})\sum_{i=1}^{n}(y_i - x_i)^2$$

As it was defined like a module, the forwarding of this class gives back the actual loss, and the backward process returns the derivative of the MSE.

III. Optimizer

In this section, we explain how we built our optimizer for the Deep Learning neural network. Our optimizer is based on the Stochastic Gradient Descent (SGD) method for optimizing the parameters of the model. We can see its constructor below :

```
optimizer = SGD(model.param(),lr,
    reduce_lr_patience,reduce_lr_factor,
    early_stop_patience)
```

The initialization of the optimizer is simple and clear. It takes as input the parameters of all the model's modules, through the `param` method. We then define a few float numbers:

- The learning rate which can also be reduced by the optimizer
- The patience which controls how many epochs we wait without improvement in the validation loss before lowering the learning rate
- The factor by which we reduce the learning rate in that case
- The patience for the early stopping

For each epoch, after the backward pass, the optimizer calls its `step` function to perform the gradient descent. Then, after the validation, if the learning is stucked in a local minima, it will lower the learning rate. To avoid starting to overfit, the optimizer can also stop the training earlier.

IV. Testing

After having implemented our framework, we were tasked with testing it with a train set and test set that we generated ourselves. The neural network, based on our framework, was composed of two input units, two output units and 3 hidden layers of 25 units. It was trained with the MSE loss. After some trial and error, the final architecture was :

- `Linear`(2 x 25)
- `TanH`
- `Linear`(25 x 25)
- `TanH`
- `Linear`(25 x 25)
- `TanH`
- `Linear`(25 x 2)
- `Sigmoid`

A. Data

The data was generated as follows: 1000 points sampled uniformly in $[0, 1]^2$, each with a label 0 if outside of the disk of radius $\frac{1}{2\pi}$ and 1 if inside. We can see it on figure 1.

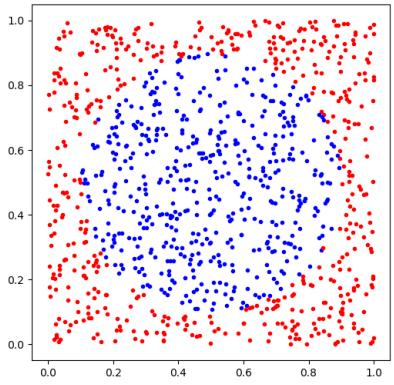

Fig. 1: Data generated, 1000 samples in blue when inside the circle of radius $\frac{1}{2\pi}$, red otherwise.

We generated 1000 samples for training, and 1000 samples for testing. Then the training set was split to have a validation set containing 20% of the points. The data is then fed in the model by mini-batches of a configurable size.

B. Results

After running the network on 1000 epochs, a mini-batch of 10 points and a learning rate of 0.1, we ended up with a final testing accuracy of 97.4%.

V. Conclusion

As seen in the following figure 2, the results are very satisfying. During our tests, we rarely needed to go up to the 1000^{th} epoch before having an early stop of the training and our testing accuracy always hovered around the 97% accuracy. The time spent implementing methods for a quite simple task makes us reflect on using already existing framework, as it can become very complex and might need to evolve and maybe be modified or adapted.

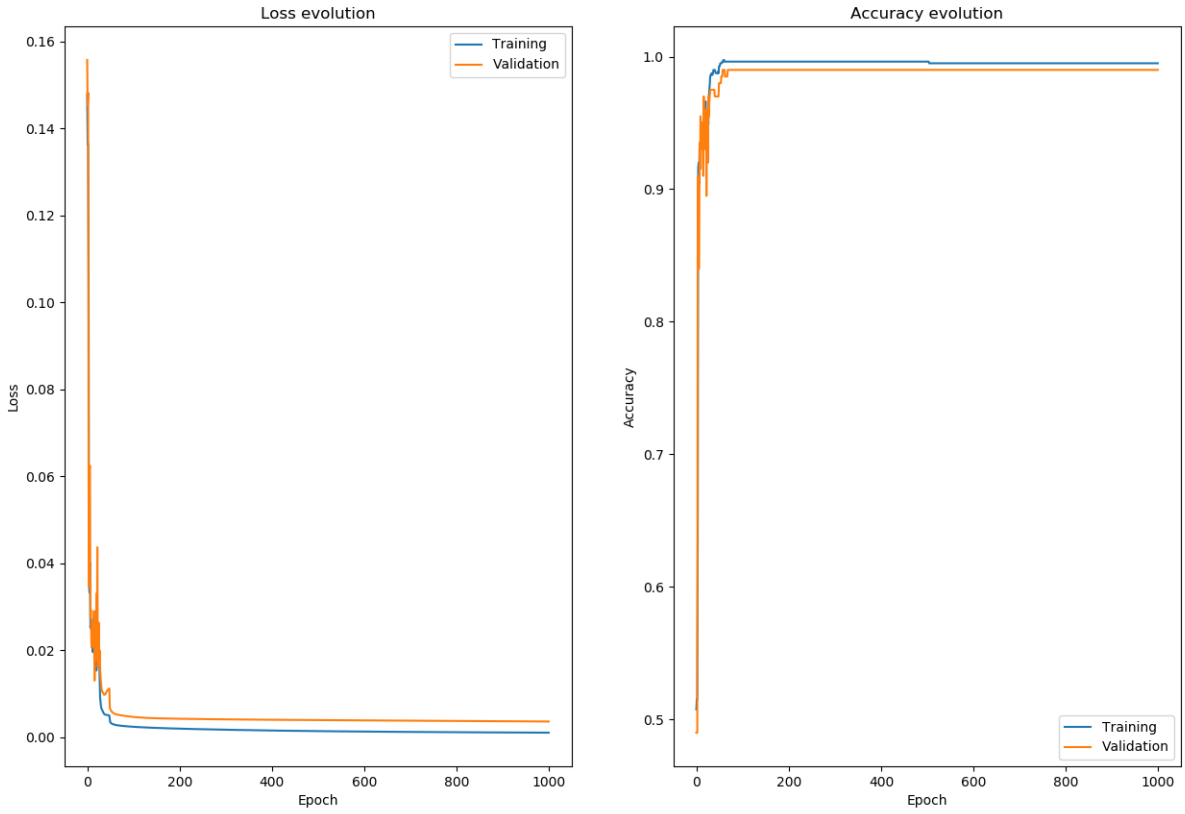

Fig. 2: Loss and accuracy evolution over the epochs.