Project 2 - Mini deep-learning framework

Emilio Fernández, Alejandro Bonell, Berk Mandiracioglu
Deep Learning, EPFL, 2020

***Abstract*—The goal of this report is to explain the development of a custom neural network framework based on Pytorch tensors from scratch. In other words, our task is to directly implement the necessary steps in the training of a neural network (feed-forward, back-propagation and parameter-optimization) without making use of the well-beloved autograd module or any related one. To check the proper functioning of our framework, we will train a simple neural network consisting of three hidden layers so that it can label whether or not a 2d-point is inside a circle.**

I. Introduction

The report has the following structure: First, in section II, the developed framework (neuralframework.py) presented and we explain how it should be used, after commenting on the "conventional" way of defining a neural network in Pytorch using torch.nn torch.autograd, so that we can distinguish the differences . Next, in section III, the framework is implemented on an executable file test.py. We describe the dataset generation, the neural network architecture and the error evolution over the training.

II. Framework documentation

A. Pytorch reference example

In order to develop the framework, we decide to use the code shown in Figure 1 as a guideline. In the example, both input data x and label data y are declared as variables, which represent nodes in a computational graph storing both data and gradients.

The model is defined with a sequential module from torch.nn which has linear layers (with different nodes) and activation functions as arguments. The loss is defined separately (in this case it is the Mean Squared Error). The network training is inside a for-loop, where several steps occur in each iteration:

- Feed-forward $\rightarrow y\_pred = model(x)$
- Loss computation $\rightarrow loss = loss\_fn(y\_pred, y)$
- Back-propagation $\rightarrow loss.backward()$
- Parameter optimization $\rightarrow data- = learn\_rate*grad$

It is worth mentioning that before the back-propagation, the function $zero\_grad$ has to be called. This function we set the gradients to zero before starting to do back-propagation because PyTorch accumulates the gradients on subsequent backward passes

```
import torch
from torch.autograd import Variable

N,D_in, H, D_out = 64,1000,100,10
x = Variable(torch.rand(N,D_in))
y = Variable(torch.rand(N,D_out),
             requires_grad=False)

model = torch.nn.Sequential(
            torch.nn.Linear(D_in,H),
            torch.nn.ReLU(),
            torch.nn.Linear(H,D_out))
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
for t in range(500):
        y_pred = model(x)
        L = loss_fn(y_pred,y)
        model.zero_grad()
        loss.backward()
        for p in model.parameters()
            p.data -= learning_rate*p.grad.data
```

Figure 1: Pytorch code example using autograd

```
import torch
import neural_framework as nf

N,D_in, H, D_out = 64,1000,100,10
x = torch.rand(N,D_in).type(DoubleTensor)
y = torch.rand(N,D_out).type(DoubleTensor)

model = nf.Sequential(
            nf.Linear(D_in,H),
            nf.ReLU(),
            nf.Linear(H,D_out),
            nf.LossMSE())

learning_rate = 1e-4
for t in range(500):
        pred = model.forward(x)
        L = model.loss(pred,y)
        model.backward()
        model.learn(learning_rate)
```

Figure 2: Corresponding code using custom framework

B. General "over the hood" idea

After having analysed the "conventional" way of defining a neural network, we select which things we want to emulate and which ones we want to change.In Figure 1, the corresponding code using neural_framework.py is shown for comparison purposes. The input data and labels are defined as tensors instead of variables given that the module autograd cannot be imported.

The framework is based on the parent class $Module$, which is composed of three methods:

- $forward \rightarrow$ computes next feed-forward data tensor
- $backward \rightarrow$ computes next backward gradient tensor
- $param() \rightarrow$ returns parameters (weights and biases)

We like the idea of the model definition using the sequential module so we opt for also including the loss definition as a module inside the model. Finally, the 4 steps of neural network training (feed-forward, loss computation, back-propagation and parameter optimization) are made more intuitive given that they each have an associated method in the sequential model: $forward(), loss(), backward(), learn()$

We now proceed to explain the details of the different modules (subclasses) in order to understand what is happening under the hood.

C. Activation functions

The framework provides with three different modules corresponding to three different activation functions: $Sigmoid()$, $Tanh()$ and $ReLU()$. These modules should always be placed after fully connected layer, so that in the forward pass, they apply the non-linearity to the transformed input: $a = \sigma(z)$. Regarding the backward pass, they compute they multiply the back-coming gradient with the activation function derivative $\sigma(z)\prime$. Given that this derivative can usually be expressed as a function of the original activation function, the result and the input of the forward-pass are stored as parameters. Finally, the method $param()$ returns an empty list given that no weights or bias are stored.

D. Fully connected layers

The module $Linear()$ allows the definition of fully connected layers with dimensions $D_{in}xD_{out}$. The weights are initialized with He initialization and the bias are initially set to zero. In the forward pass, the input data is stored as an attribute and it is linearly transformed ($z = XW + b$). In the backward pass, three gradients are computed: $\frac{\partial L}{\partial x}, \frac{\partial L}{\partial w}, \frac{\partial L}{\partial b}$. The loss gradient with respect the linear input is returned for continuing the back-propagation, whereas the loss gradient with respect weights and biases are stored as attributes. These attributes are used in the $optimise()$ method where the parameters are updated ($w = w - \eta * \frac{\partial L}{\partial w}$). Finally, the method $param()$ returns a list of all parameters with their corresponding gradient

E. Loss computation

In the forward pass, the module $LossMSE()$ computes the Mean Squared Error with respect the expected label. In the backward pass, the derivative of the loss is computed and returned so that the back-propagation can start.

$$L_{MSE} = \frac{1}{N}\sum_{n=1}^{N}(y_{predicted} - y_{true})^2$$

F. Sequential model

The most important subclass in the framework is probably the $Sequential()$ module given that it allows to combine multiple layers when defining the neural network architecture. The layers (modules) are stored in a attribute list which is used as iterable when calling the methods $forward()$ (direct order) and $backward()$ (reversed order). The method $param()$ also iterates over all modules for getting all the parameters and is able to print them. The method $learn()$ calls the method $optimise()$ of every linear layer for updating the parameters and The method $loss()$ calls the forward method of the last argument in the sequential module for computing the loss.Finally, the method $error()$, transforms the predictions in either ones or zeros so that they can be compared with the true labels. The predictions which do not match are summed up and divided by the total number of predictions, so that the error can be returned.

III. Framework implementation

After having specified the working principles of our deep-learning framework, it is time to see if it really works. For that purpose, we train a small neural network in an executable file test.py which imports neural_framework.py and tools.py (containing extra functions for the proper functioning of the executable).

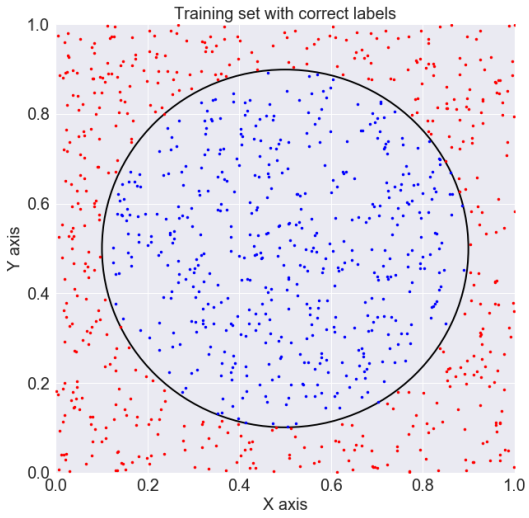

Figure 3: 1000 uniformly distributed points

A. General description of the classification task

The neural network is composed of three hidden layers (two ReLUs and one Tanh) with 25 nodes each and it must classify two dimensional points based on whether the point is inside a circle as shown in Figure 3. The data set is generated using a random uniform selection of N=1000 points between the range [0,1] for each dimension. The labels are assigned by comparing the radius of each point in polar coordinates with the circle radius Therefore, the neural network has two input units corresponding to the horizontal and vertical coordinates of the points, and two output units corresponding to the probability of being inside or outside the circle, respectively.

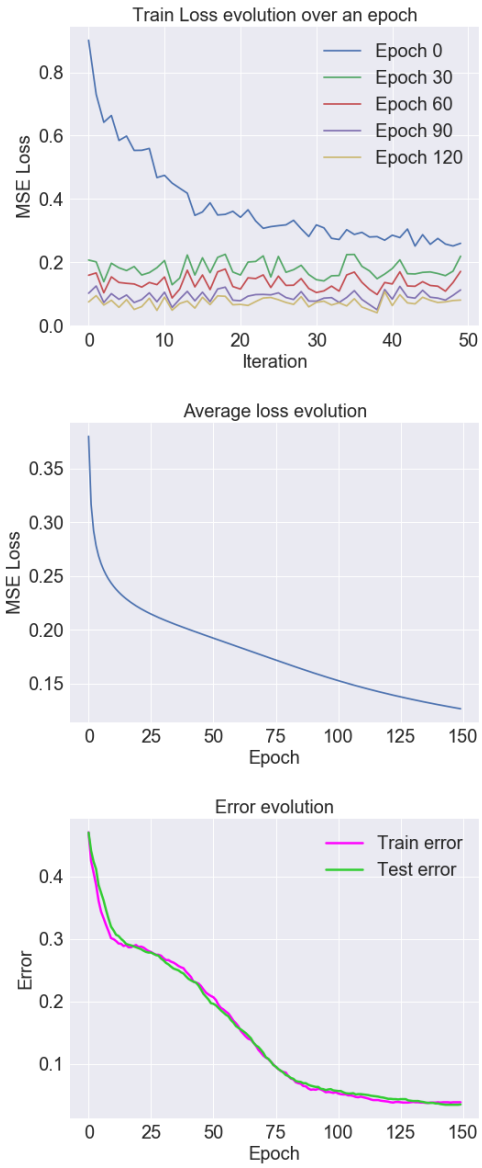

Figure 4: Loss and error evolution

B. Neural network training

The framework is implemented so as to optimise the model with gradient descent as shown at the beginning of the report in Figure 2. However, in this case, we divide the input data in batches of 20 samples in order to optimise the parameters with Mini-Batch Gradient Descent. In other words, the neural network is only presented with 20 samples at every iteration instead of with the whole data set.

Although this makes the learning more irregular and inefficient (gradient is only being computed with a limited number of samples), the main advantage is that the final trained model is much more robust because the noisiness of the picked samples prevent the model from converging to saddle points or local minima. This learning process is repeated for 150 epochs and we can observe in Figure 4 that the MSE loss is reduced after every epoch even if the iterations in the same epoch are irregular. After 150 epochs and 50 iterations for every epoch, we obtain a train error of 3,85% and, to our surprise, a test error of 3,50%. This means that there is not over-fitting because the model performs even better on the test set than on the training set.

Finally, in Figure 5 we can see the predicted labels for the test set and we check that most of the points are correctly classified.

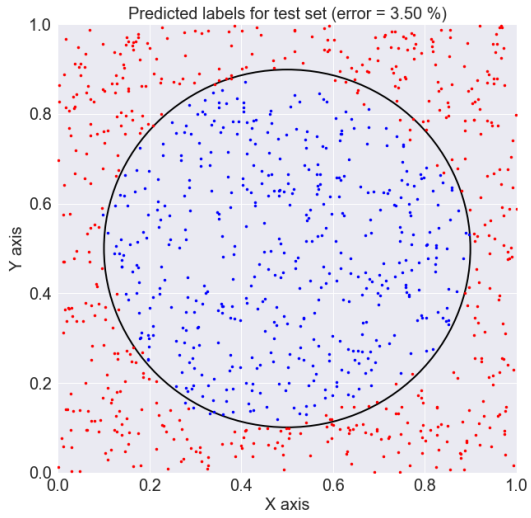

Figure 5: Final prediction for test set

IV. Conclusions

In conclusion, we have sucesfully developed a basic deep learning framework and a simple executable which allows to play with different activations functions and newtork architectures. There is room for improvement given that other types of classes as well as more options can be incorporated to the framework, but in general we are happy and proud of our work because it takes advantage of the potential behind tensors without using the magic of $autograd()$