Project 2: Mini Deep Learning Framework in Python

Johan Claudius Jean-Paul Félisaz, Lorenzo Robert Jacqueroud, Amir Rezaie, *EPFL Lausanne, Switzerland*

***Abstract*—In this project, a new deep learning framework was developed capable of building fully-connected neural networks in Python language using Tensor operations of the PyTorch library. To test the implemented framework, a binary classification task was solved using a synthetic dataset.**

I. Introduction

The aim of this project was to develop a deep learning framework capable of building fully-connected networks (FCN) in Python using only Tensor operations implemented in the PyTorch package. The main python classes to build a FCN were defined in a python module called "NN.py", which includes: class "Module" that is the base class for other classes, class "Linear" in which one fully-connected layer can be defined, class "ReLU", and "Tanh" as an activation, class "Sequential", which contains a sequence of the instances of classes "Linear" and "ReLU" or "Tanh", and the class "LossMSE" to compute the mean squared loss.

To test the implemented framework, a dataset of two classes were generated; and then using the defined loss module, and splitting the data into train and test, the performance of a FCN was evaluated.

In what follows, the dataset is firstly introduced, secondly, two implemented architectures and optimization scheme are detailed. Finally, the performance of the models is investigated.

II. How to use the developed framework

In this section, necessary steps and the syntax to build and train a FCN in the developed framework are briefly explained.

A. Build a FCN

To create a FCN in the developed framework, an instance of the class "Sequenctial" needs to be created as shown in Figure 1, which has 2 input unit, 3 hidden layers each with 25 units and two output units. In the developed framework, two activation functions "ReLU" and "Tanh" can be used. As shown in Figure 1, to apply an activation function, an instance of the classes "ReLU" or "Tanh" must be created right after a "Linear" layer.

B. Model optimizer

In this framework, to train a model, the mini-batch SGD was used. After creating a model such as the one shown in 1, the learning rate (lr) can be defined as illustrated in Figure 2.

C. Loss function

In the current version of the framework, the Mean Squared Error (MSE) loss is existed. To define a criterion, for the evaluating and training a model, an instance of the class "LossMSE" must be created (see Figure 3).

Figure 1: Python code for building a FCN in the developed deep learning framework

```
model = Sequential(
                    Linear(2, 25),
                    ReLU(),
                    Linear(25, 25),
                    ReLU(),
                    Linear(25, 25),
                    ReLU(),
                    Linear(25, 2)
                    )
```

Figure 2: Defining the learning rate used in the mini-batch SGD.

```
model.optimizer['eta'] = 1e-4
```

D. Forward and Backward pass

To perform a forward pass, a batch of the design tensor **X_batch** of the size (batch_size $\times$ number of features $\times$ 1) must be passed as an argument to the instance "model", as shown in Figure 4.

To compute the gradients of the loss function w.r.t weights and parameters, the method "grad" of the instance "model" must be called and a batch of the target tensor **y_batch** of the size (batch_size $\times$ number of classes $\times$ 1) must be passed to this method (see Figure 4).

Finally, to update weights and perform a backward pass, the implemented method "backward" of the model instance must be called (see Figure 4).

E. Accessing weights and biases and their derivatives

All parameters of the model and their derivatives can be accessed through an attribute of the model instance called "layers". This attribute is a dictionary containing FC layers and activation functions. As an example, the code "model.layers" was run for the model shown in Figure 1 and a part of the output, which is a dictionary is reported in Figure 5. To access to the parameters and their derivatives of, for example, the first FC layer, lines [7-11] of the code shown in Figure 5 must be run.

III. Problem definition

To verify and test the implemented deep learning framework, a binary classification task was solved. The synthetic dataset for this problem is explained below. To generate the training and test dataset, two sets each including 1000 points with two dimensions $\{x_1, x_2\}$ was sampled from a uniform

Figure 3: Defining the criterion to optimize.

```
criterion = LossMSE()
```

Figure 4: Forward and backward pass

```
# forward pass
pred = model(X_batch)
# gradient step and backward pass
model.grad(y_batch)
model.backward()
```

Figure 5: Accessing model layers

```
model.layers
 OUTPUT:
 {'fc_1': Fully-connected layer: in_features:2,
    out_features:25,
 'act_1': ReLU activation,
 'fc_2': Fully-connected layer: in_features:25,
    out_features:25,
 ...}
 model.layers['fc_1']._weights
 model.layers['fc_1']._bias
 # after running model.grad(y_batch)
 model.layers['fc_1']._d_bias
 model.layers['fc_1']._d_weights
```

distribution $U(0,1)$. Then, these points were labeled according to the following rule:

$$y = \begin{cases} 0 & 1/\sqrt{2\pi} < \|\boldsymbol{x}\|_2 < 1 \\ 1 & \text{otherwise} \end{cases}$$

where, y is the class label. An example of generated training samples and the corresponding class regions are shown in Figure 6. The area of the blue (class 0) and red (class 1) regions occupies around 66% and 34% of the domain area, respectively.

IV. Modeling

A. Architecture

The base architecture used in this study was the one shown in Figure 1 and described in section II-A.

However, the activation function was considered as a variable parameter to investigate its effect on the model performance (see section IV-D.)

B. Optimization scheme

As mentioned in section II-B, the mini-batch SGD is the only implemented optimizer in this framework to optimize the parameters of a model based on the MSE loss function. To train models, the number of epochs was fixed to 500 epochs.

The performance of trained models was compared by changing hyper-parameters lr and batch size that are detailed in section IV-D.

C. Evaluation metric

One common evaluation metric for a binary classification task is "Accuracy", which is used in this study and defined below:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

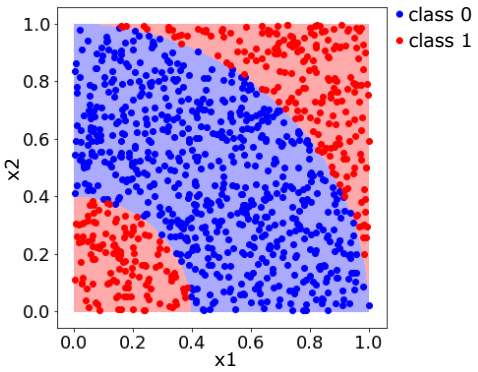
where, TP and TN are true positive and true negative predictions, respectively, and FP and FN are false positive and false negative predictions, respectively.

D. Experiments

In this study, the influence of the following parameters on the performance of a model was explored:

1) **Activation function**: Two functions, "ReLU" and "Tanh" were used as the activation function for all hidden layers.

Figure 6: Illustration of a generated training set. The blue regions/markers belong to the class 0 and the red regions/-markers belong to the class 1. The dataset was generated by fixing the seed number, i.e. torch.manual_seed(42).



2) **Learning rate**: The lr was chosen from the set $\{1e-2, 1e-3, 1e-4, 1e-5, 1e-6\}$.
3) **Batch size**: The batch size was selected from the set $\{1, 5, 10, 20, 50, 100, 200\}$.

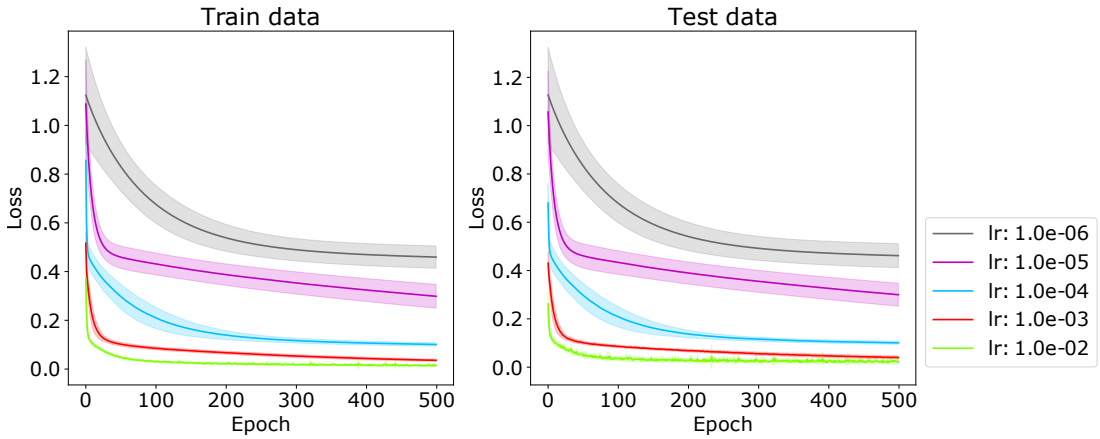
The combinations of these parameters resulted in 70 different models. Moreover, to obtain the standard deviation of accuracy and loss, each model was trained 10 times by generating new train and test datasets.

V. Results and Discussion

For brevity, in this section, only plots of the models with ReLU activation function are illustrated. The results of models with Tanh activation are summarized in Table I.

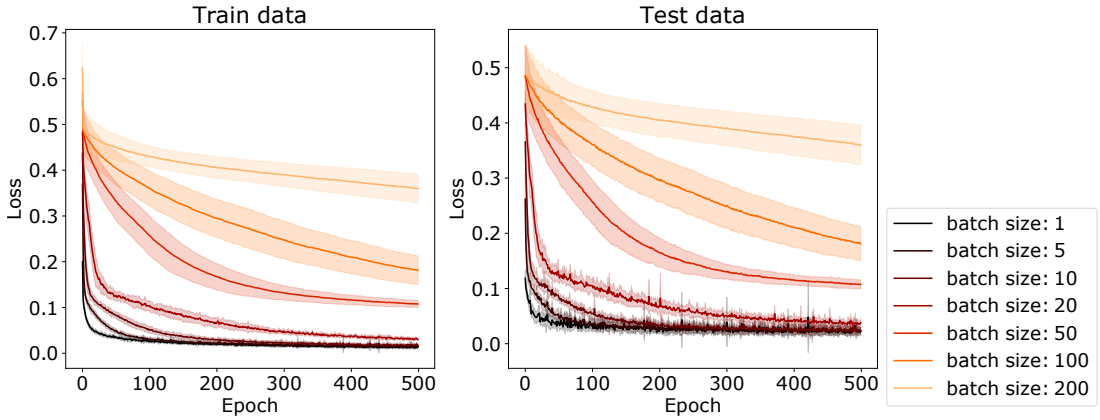
Figure 7 shows plots of the MSE loss evaluated over the train and test data vs. epoch for the model with the batch size equal to 5. In this figure, lines show the mean values of ten rounds of training, and the shaded areas show one standard deviation around the mean. It can be seen that as expected by increasing the learning rate the loss over the train and test data decreases with higher rate; and for this batch size, the learning rate of 0.01 resulted in a model with the lowest averaged (over 10 rounds of training) test loss. The same trend was also observed for the models with other batch sizes. The results of the loss and accuracy values, their standard deviation (std) and the learning rate, which resulted in the lowest averaged test loss, are summarized in Table I.

Figure 7: Plots of loss over train/test data vs. epoch for the model trained with batch size = 5 and ReLU activation



After tuning the learning rate for each model that was trained with a specified batch size, the effect of the batch size on loss values can be evaluated. Figure 8 plots the train/test loss vs. epoch for models trained with different batch sizes and ReLU activation. It is observed that by decreasing the batch size, the model converges to a smaller averaged loss. The performance of the trained models (both with ReLU and Tanh activation) is almost similar for batch sizes 1 and 5 (see Table I). Between all these models, the model trained with the batch size of 5 and the ReLU activation had the lowest averaged test loss of 0.0187 with the std of 0.0043, and the highest accuracy of 98.88% with the std of 0.27%.

Figure 8: Plots of loss over train/test data vs. epoch for all batch sizes and ReLU activation, after tuning the lr



Batch size	lr	Train loss		Test loss		Train Accuracy		Test Accuracy	
		ReLU act.	**Tanh act.**	**ReLU act.**	**Tanh act.**	**ReLU act.**	**Tanh act.**	**ReLU act.**	**Tanh act.**
1	0.01	0.0160 ± 0.0027	0.0171±0.0025	0.0189 ± 0.0047	0.0191± 0.0039	98.82± 0.19	98.78± 0.17	98.82±0.29	98.78±0.24
5	0.01	0.0148± 0.0033	0.0164± 0.0032	**0.0187** ± 0.0043	0.0198±0.0044	98.78±0.27	98.78± 0.27	**98.88**±0.27	98.78±0.36
10	0.01	0.0174± 0.0041	0.0269± 0.0099	0.0220± 0.0058	0.0292± 0.0053	98.61± 0.29	98.56± 0.45	98.61± 0.44	98.56±0.32
20	0.01	0.0987± 0.0028	0.0708± 0.0148	0.0339 ± 0.0037	0.0704± 0.0114	98.39± 0.22	96.63± 0.91	98.39±0.36	96.63±0.83
50	0.001	0.1077 ±0.0078	0.1290±0.0104	0.1071 ±0.0076	0.1270±0.0073	96.28± 0.63	94.89± 1.25	96.28±0.79	94.89±0.89
100	0.001	0.1815± 0.0311	0.2748±0.0628	0.1812± 0.0309	0.2736±0.0595	90.48± 3.67	81.74± 2.91	90.48±3.62	81.74±4.00
200	0.001	0.3600 ± 0.0305	0.3866±0.0308	0.3598± 0.0363	0.3865± 0.0286	72.76± 4.06	75.93± 6.36	72.76±5.87	75.93±6.45

Table I: Performance of models over train and test data