EE559 Project 2: Mini Deep-Learning Framework

Lingjing Kong, Jiahua Wu, Phan Huy Thong
Ecole Polytechnique Federale de Lausanne, Switzerland

***Abstract*—In this project, we implemented a mini deep learning framework with the sufficient flexibility and basic functionalities of PyTorch framework. Based on our framework, a simple multiple layer perceptron (MLP) is built and tested on a binary classification task. We evaluated our framework by comparing its results and running time against those of the PyTorch framework.**

I. Overview

The aim of the project is to design from scratch a deep learning framework using only PyTorchs tensor operations and the standard math library and in particular without using autograd or the neural-network modules. The framework is proved usable through a binary classification task. We shall start by presenting the essential modules along with mathematical definitions and our implementation and then we test our framework against the PyTorch framework on a artificial classification task. The report will be ended by a conclusion. In what follows, ℓ denotes the loss function. $\mathbf{x}^{(l)}$ denotes the output of the activation function at layer l, i.e. the input to the linear layer $l+1$; $s^{(l)}$ denotes the output of the linear layer $l-1$, i.e. the input of the activation function on layer l,

II. Modules

A. Linear Layer

Linear layer is the skeleton of a fully connected neural network. The layer performs a linear transformation on output from the previous activation function layer and send it to the next one.

1) Forward Pass: The linear transformation can be represented by a matrix multiplication between the input and the weights and an addition of the resulting term with the bias:

$$\mathbf{s}^{(l)} = \mathbf{W}^{(l)}\ \mathbf{x}^{(l-1)} + \mathbf{b}^{(l-1)}$$

2) Initialization: A proper initialization of the weights and bias is a remedy to vanish gradient problem. In our framework, we adopt the initialization method introduced in the lecture:

$$\mathbf{W} \sim \mathrm{U}(-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}) \quad \mathbf{b} \sim \mathrm{U}(-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}})$$

where U represents uniform distribution and n represents the size of the input vector.

3) Backward Pass: The gradient of the loss function w.r.t the weights and the bias can be calculated as follows:

$$\frac{\partial \ell}{\partial \mathbf{W}^{(l)}} = \frac{\partial \ell}{\partial \mathbf{s}^{(l)}} \mathbf{x}^{(l-1)^T} \quad \frac{\partial \ell}{\partial \mathbf{b}^{(l)}} = \frac{\partial \ell}{\partial \mathbf{s}^{(l)}}$$

$$\frac{\partial \ell}{\partial \mathbf{x}^{(l-1)}} = \mathbf{W}^{(l)T} \frac{\partial \ell}{\partial \mathbf{s}^{(l)}}$$

As the formula show, the input is used in the calculation of the gradients so in our implementation it is stored in an attribute of the class.

4) Parameters: For use in training procedure, the parameters will be returned by a list of tuples:

$$(\mathbf{W}, \frac{\mathrm{d}\ell}{\mathrm{d}\mathbf{W}}), (\mathbf{b}, \frac{\mathrm{d}\ell}{\mathrm{d}\mathbf{b}})$$

B. Relu Layer

The rectified linear unit is a commonly used activation function for neural networks (NN) at present which has a frequent presence in various structures.

1) Forward Pass: The layer receives the output of a linear layer and applies a non-linear transformation defined as:

$$\mathbf{x}_{ij}^{(l)} = \mathrm{ReLU}(\mathbf{s}_{ij}^{(l)}) = \begin{cases} \mathbf{s}_{ij}^{(l)}, & \text{if } \mathbf{s}_{ij}^{(l)} > 0 \\ 0\ , & \text{otherwise} \end{cases} \tag{1}$$

2) Backward Pass: The gradient of the loss function w.r.t to the input is given as:

$$(\frac{\partial \mathbf{x}^{(l)}}{\partial \mathbf{s}^{(l)}})_{ij} = \begin{cases} 1 \text{ if } \mathbf{s}_{ij}^{(l)} > 0 \\ 0 \text{ otherwise} \end{cases}$$

$$\frac{\partial \ell}{\partial \mathbf{s}^{(l)}} = \frac{\partial \ell}{\partial \mathbf{x}^{(l)}} \odot \frac{\partial \mathbf{x}^{(l)}}{\partial \mathbf{s}^{(l)}}$$

where $\odot$ means the Hadaman multiplication. As one can notice, the input is needed for computation of the gradient so it is stored as an attribute of the class.

C. Tanh Layer

The tanh function is a sigmoid function takes value from $(-1, 1)$ and it is used as non-linear activation function.

1) Forward Pass: The function is defined as:

$$\mathbf{x}_{ij}^{(l)} = \tanh(\mathbf{s}_{ij}^{(l)}) = \frac{\exp(\mathbf{s}_{ij}^{(l)}) - \exp(-\mathbf{s}_{ij}^{(l)})}{\exp(\mathbf{s}_{ij}^{(l)}) + \exp(-\mathbf{s}_{ij}^{(l)})}$$

2) Backward Pass: The gradient w.r.t the input $\mathbf{s}$ is calculated as:

$$(\frac{\partial \mathbf{x}^{(l)}}{\partial \mathbf{s}^{(l)}})_{ij} = \frac{4}{(\exp(\mathbf{s}_{ij}^{(l)}) + \exp(-\mathbf{s}_{ij}^{(l)}))^2}$$

$$\frac{\partial \ell}{\partial \mathbf{s}^{(l)}} = \frac{\partial \ell}{\partial \mathbf{x}^{(l)}} \odot \frac{\partial \mathbf{x}^{(l)}}{\partial \mathbf{s}^{(l)}}$$

where $\odot$ refers to the element-wise product. Similar to other layers, the input is equally needed for calculation of the gradient and therefore it is saved as an attribute of the layer class.

D. Sequential Module

Evidently the backward propagation scheme requires a sequential data propagation between different layers. Therefore, we construct a module type `Sequential` which takes as argument a list of layers, transfers sequentially the data through the method `forward_pass` and propagates the error inversely through `backward_pass`.

1) Forward Pass: During the forward pass, the module iterates successively through the layers, executes each one's `forward_pass` method and at the same time transfers the output of one layer to the next one as input.

2) Backward Pass: The module iterates inversely through the list of layers, calls each one's `backward_pass` method to calculate their own gradient and passes from one layer to the next one.

E. MSE Loss

The MSE loss is a classical loss function for regression task which quantifies the euclidean distance between the prediction and the true value. Here N denotes the number of samples, $\mathbf{y}_i$ denotes the model prediction on the i the sample and $\mathbf{y}_i^*$ refers to the label.

1) Definition: This loss function is defined as follows:

$$\ell = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{y}_i - \mathbf{y}_i{}^*)^2.$$

2) Backward Pass:

$$\frac{\partial \ell}{\partial \mathbf{y}_i} = \frac{2}{N}(\mathbf{y}_i - \mathbf{y}_i^*)$$

F. Mini-batch SGD

In the SGD algorithm, the parameters are updated each time after a random data point is passed through. In our implementation, we extend the algorithm to a mini-batch way, i.e. we process a batch of data points instead of one at the each pass. This variant of gradient descent induces computational speed-up compared with the gradient descent since it cuts down the number of copies of parameters transferred to the cache during training, which effectively saves the communication cost. Additionally, the gradient it calculates has less variance compared to the one obtained by SGD.

III. Performance Testing

For the sake of the assessment of our framework, we have compared its performance to the PyTorch on a simple binary classification task. The comparison is in two aspects: accuracy and speed.

A. Data Generation

Two datasets, each comprised of 1000 data points uniformly sampled in $[0,1]^2$, are separately generated for training and testing. The points inside a disk of radius $\frac{1}{\sqrt{2\pi}}$ are labeled as 1 and those outside are labeled as zero as shown in Figure1.

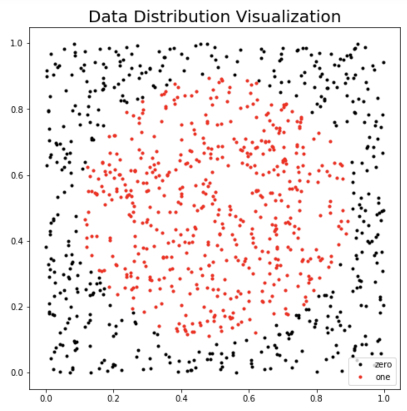


Fig. 1. Distribution of data labels: the red points have label 1; the black ones have label 0

B. Network Structure

We choose an architecture according to the guidance in the project description. Namely, a network with 2 input units, three hidden layers comprised of 25 units with relu activation function and two output units. More precisely, in form of a code:

```
Sequential([Linear(2, 25), ReLu(),
            Linear(25, 25), ReLu(),
            Linear(25, 25), Tanh(),
            Linear(25, 2), Tanh()])
```

C. Test Setting

Tests were conducted under the setting of parameters shown in Table I.

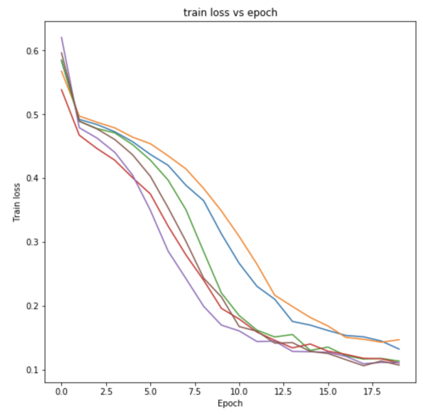
Parameter	Value
Batch Size	10
Learning Rate	0.01
Epochs	20

TABLE I
Test Parameters

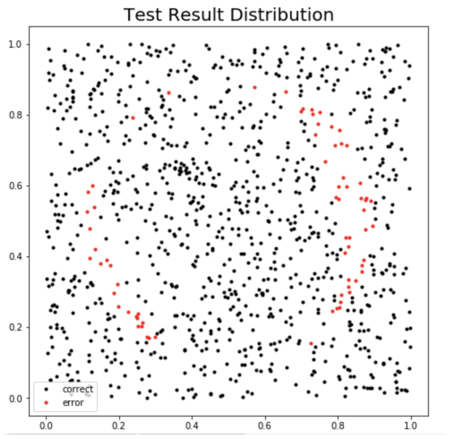
D. Comparison against PyTorch

As a reference, the same network structure is built in PyTorch with the same net structure, optimizer and loss function as the model built in our framework. In addition to the experiment mentioned in project description, the two frameworks are also tested on large datasets in order to gain a better view of the performance regarding the speed.

1) Performance in standard test: Firstly we compare the performance of our framework with that of PyTorch through the binary classification problem proposed in the project description. The learning curve of the network as well as the distribution of mis-classified data points in our framework are shown in Figure 2. A similar convergence behaviour of training can be observed in the curves shown in the former figure while from the latter one can notice that the mis-classified data points are located around the true boundary, which implies that the network is able to construct a criterion close to the true boundary and classifies the data points based on it. To obtain a reliable result, the test is run 10 times and the result is summarized in Table II.



(a) Learning curves of 6 experiments in the mini framework.



(b) Distribution of mis-classified data points. In line with experience, these points are located near the boundary of the positive label disk where the data points are the least distinguishable

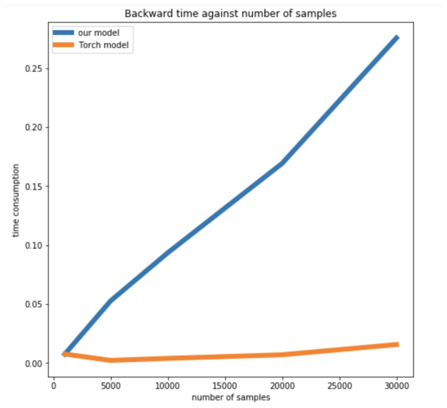
Fig. 2. Learning Outcomes

	Our framework	PyTorch
Initialization/ms	0.987	1.546
Forward(1000 samples)/ms	6.52	1.482
Backward(1000 samples)/ms	7.676	7.728
Test accuracy	96.8	94.7

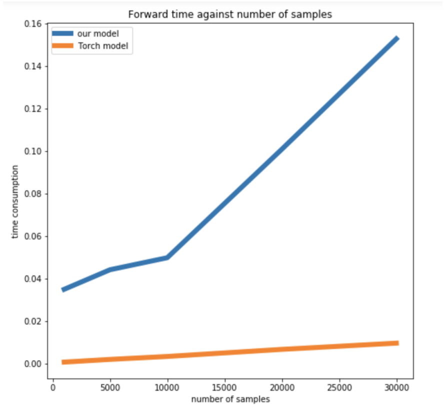
TABLE II
COMPARISON WITH PYTORCH (AVERAGED TIME AND ACCURACY OF 10 EXPERIMENTS)

As revealed from the experiment, the test accuracy of the same network structure does not show much difference in the two frameworks. However, our framework is more efficient in initialization but much slower in forward pass. The faster initialization may be due to the simplicity of our model: we do not have complex data structures used in PyTorch. The latter may be due to the memory allocation: in the process of forward pass, the input data has to be cloned in each layer for later use during backward pass whereas PyTorch deals with the reuse of input data in a more sophisticated way.

2) Performance on large dataset: We also conduct the same experiment on larger dataset and the results are shown in Figure 3.



(a)



(b)

Fig. 3. Comparison of the speed between the mini framework and PyTorch on two main procedures of NN training.

Obviously, our mini-framework built upon straightforward implementation is not suitable for large scale training compared with PyTorch whose algorithm is implemented in a more optimal manner.

IV. CONCLUSION

In this project we build a rudimentary NN framework which includes sequential model, linear layer, ReLU layer, Tanh layer, MSEloss and mini-batch training. Admittedly we did not spend much time in developing blocks such as dropout, batch normalization and other activation functions, we dedicated our time to improving the readability and documentation of our code, hoping to make it as user-friendly as PyTorch code.