PROJECT 2: MINI DEEP-LEARNING FRAMEWORK

DENG Zhantao
zhantao.deng@epfl.ch

LI Junze
junze.li@epfl.ch

ABSTRACT

In this project, we design a mini 'deep learning framework' using only PyTorch's tensor operations and the standard math library. The framework has similar APIs with PyTorch functions. It contains the following modules: Tanh/ReLU activation function, batch normalization, linear layer, MSE loss function, SGD optimizer, and sequential. We build a simple network based on this mini framework and the dataset is generated manually in 2D space. Finally, we compare the network performance between our framework and PyTorch.

1 Framework description

1.1 Weight matrix

The cornerstone of our framework is the learnable parameter, including the weight, bias and gradient. To properly organize these parameters, we define a weight module, which includes value, gradient, initialization (Gaussian distribution) and reset (both the value and gradient). It is used by all modules having trainable parameters.

1.2 Activation function module

We implement two activation functions, Tanh and ReLU. They have the simplest structure in the framework, i.e. only forward and backward pass, since there is no parameter to be determined during training. The Tanh function and its gradient are shown in 1 and the ReLU function and its gradient are shown in 2.

$$y = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \quad \frac{\partial y}{\partial x} = 1 - \tanh^2(x) \tag{1}$$

$$y = ReLU(x) = \max(x, 0), \quad \frac{\partial y}{\partial x} = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{2}$$

1.3 Linear layer module

The linear layer module is used to achieve the fully connected layer in neural networks. The linear function can be denoted as,

$$\mathbf{y} = \mathbf{W}\mathbf{X} + \mathbf{b} \tag{3}$$

where **W** denotes the weight matrix with the size (input data dimension×output data dimension), **X** denotes the input tensor with the size (batchsize×input data dimension), and **b** denotes the bias vector with the size (1×output data dimension).

The gradient of **W**, **b** and **X** are as following,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \mathbf{X}^T \frac{\partial \mathcal{L}}{\partial \mathbf{y}}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{X}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \mathbf{W}^T, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{b}} = \mathbf{1}^T \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \tag{4}$$

where $\mathcal{L}$ denotes the loss value.

1.4 MSE loss module

The mean square error L is defined as,

$$L = \frac{1}{N}\sum_{i,j}(\hat{y}_{i,j} - y_{i,j})^2, \quad \frac{\partial L}{\partial \hat{y}_{i,j}} = \frac{2}{N}(\hat{y}_{i,j} - y_{i,j}) \tag{5}$$

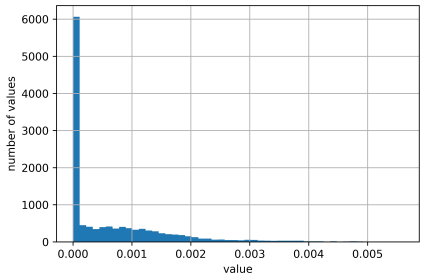
where N is label set size, i denotes the ith data in a batch, j denotes the jth data dimension, y denotes the ground truth label and $\hat{y}$ denotes the predicted label. For the backward propagation, we only need the gradient of the predicted labels as shown in 5. This module does not have any parameters that should be determined in the training process, so the weight module is not needed.

1.5 SGD optimizer module

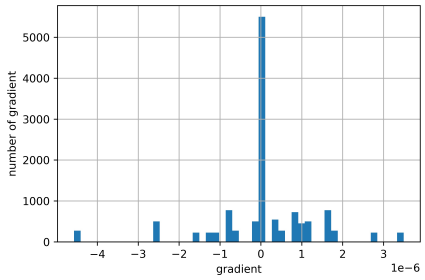
We use stochastic gradient descent (SGD) to optimize the model. This module has a different structure, i.e it only contains zeros_grad() to reset gradient for all parameters, step() to update all parameters and a initialization method. The SGD module can be initialized by passing it the trainable parameters and setting the learning rate (default value is 0.01), which is same as PyTorch.

1.6 Batch normalization module

Batch normalization is a technique to reduce the number of epochs to converge, to improve performance and stability of neural networks [1]. In our case, we observe that, without batch normalization, the input feature of deep layers will be dominated by the bias of the previous layer and the gradient of shallow layers will vanish, as shown in Figure1, leading to very slow convergence and poor performance.



(a) The distribution of input feature of the third layer



(b) The distribution of gradient of the second layer

Figure 1: Meaningless feature and vanishing gradients.

1.7 Sequential module

This module implements the sequential method to create a network.

The structure of sequential module

```
class sequential:
    def __init__(self, *structure):
    def forward(self, x):
    def backward(self, grad):
    def parameters(self):
    def test(self):
    def train(self):
```

The forward and backward functions enable tensor pass in forward and backward directions. The parameter function saves all trainable parameters of the network, but currently, only linear layer and BN layer are supported. The train and test functions change the mode of BN layer.

2 Experiment

We test our framework on the generated dataset and compare its performance with the PyTorch framework.

2.1 Data generation and network structure

We generate a dataset of 1000 points sampled uniformly in $[0, 1]^2$. If a point is in the circle with the center $(0.5, 0.5)$ and radius $1/\sqrt{2\pi}$, the corresponding label is 1 otherwise 0. We build the following network architecture and train it with the generated dataset.

The structure of network

```
model = sequential(
        Linear(input_size = 2, output_size = 25),
        ReLU(),
        batchNormalization(batchSize, input_size = 25),
        Linear(input_size = 25, output_size = 25),
        ReLU(),
        batchNormalization(batchSize, input_size = 25),
        Linear(input_size = 25, output_size = 25),
        ReLU(),
        batchNormalization(batchSize, input_size = 25),
        Linear(input_size = 25, output_size = 2))
```

2.2 Results

We run 20 rounds in total and each round contains 50 epochs. The learning rate is 0.06 and batch size is 500. We visualize the loss curve in training process and the accuracy curves on train/test set in Figure 2. We also build a same network architecture based on PyTorch framework and compare its performance with our Mini-framework's in Table 1.

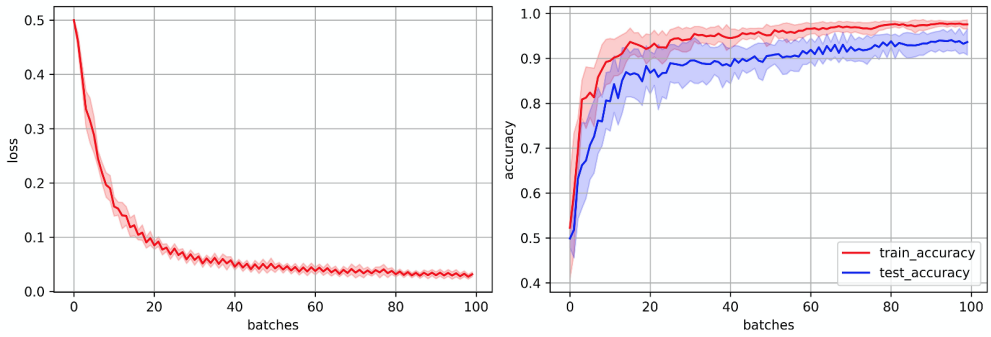


Figure 2: The train loss curve and train/test accuracy curve

Table 1: The comparison between our framework and PyTorch

Framework	Accuracy	Std
Mini-framework	93.64%	2.87%
PyTorch	96.04%	0.96%

We can find that the average accuracy of PyTorch is around 2.5% higher than Mini-framework. The standard deviation of PyTorch is lower than that of Mini-framework, which indicates that the Mini-framework is less robust.

3 Conclusion

Although the average accuracy of Mini-framework is slightly lower than PyTorch's, the overall performance of Mini-framework is still decent. By observing the standard deviation of accuracy, we can find that one of 20 rounds could get the similar accuracy of PyTorch (93.64%+2.87%), which means we should improve the stability and robustness of our framework in future work.

References

[1] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.