Implementing from Scratch A Mini Deep-Learning Framework

Tao Sun Wenlong Deng Yaxiong Luo
{tao.sun, wenlong.deng, yaxiong.luo}@epfl.ch

***Abstract*— The objective of this project is to implement a mini deep learning framework using only PyTorch tensor operations and the standard math library, hence in particular without using Autograd or the neural-network modules. Our framework consists of several modules including Linear (fully connected layer), ReLu, Tanh, Sequential for combining modules and LossMSE for computing MSE loss. For each module, forward pass and backward propagation are implemented. Mini-batch SGD is applied for optimization. A simple 2-class classification problem is used to test the functionality of our framework.**

I. Framework Implementation

This part introduces the main modules and functions we have implemented in our framework, including their structures, mathematical principles behind them, and how they are able to function together.

In following sections, $\boldsymbol{X}$ is defined as the input matrix of one layer, $\boldsymbol{Y}$ is defined as the output matrix of one layer, and $\boldsymbol{L}$ is defined as the final loss matrix of the network.

A. Linear Module

The linear module takes three arguments as input, dimension of input, $\boldsymbol{n}$, dimension of output, $\boldsymbol{m}$, and a boolean sign determining to involve bias or not.

1) Forward Propagation: The high-level reasoning in the neural network is done via fully connected layers. Neurons in a fully connected layer have connections to all activation functions in the previous layer, as seen in regular neural networks. Their activation functions can hence be computed with a matrix multiplication followed by a bias offset:

$$\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{W} + \boldsymbol{b}$$

2) Initialization: The weight $\boldsymbol{W}$ and the bias $\boldsymbol{b}$ need to be properly initialized. For the fully connection, the gradient vanishes exponentially with the depth if the weight $\boldsymbol{W}$ are ill-conditioned. The weights and bias, if involved, are initialized as following:

$$\boldsymbol{W} \sim \boldsymbol{U}\left(-\frac{1}{\sqrt{\boldsymbol{n}}}, \frac{1}{\sqrt{\boldsymbol{n}}}\right) \qquad \boldsymbol{b} \sim \boldsymbol{U}\left(-\frac{1}{\sqrt{\boldsymbol{n}}}, \frac{1}{\sqrt{\boldsymbol{n}}}\right)$$

where $\boldsymbol{U}$ is the uniform distribution, and $\boldsymbol{n}$ is the dimension of input features.

*This is report for the 2nd mini-project of the course Deep Learning at EPFL 2018 Spring given by Prof. François Fleuret.

3) Backward Propagation: With the consideration of matrix dimensions, gradients with respect to (w.r.t.) parameters and input can be calculated as following:

$$\frac{d\boldsymbol{L}}{d\boldsymbol{W}} = \boldsymbol{X}^T \frac{d\boldsymbol{L}}{d\boldsymbol{Y}} \qquad \frac{d\boldsymbol{L}}{d\boldsymbol{b}} = \left(\frac{d\boldsymbol{L}}{d\boldsymbol{Y}}\right)^T \mathbb{1}_n \qquad \frac{d\boldsymbol{L}}{d\boldsymbol{X}} = \frac{d\boldsymbol{L}}{d\boldsymbol{Y}} \boldsymbol{W}^T$$

The input of this layer is required in the gradient calculation, thus, need to be kept as a variable inside the module.

4) Parameters: The parameters of the Linear Layer is stored in a list of tuples:

$$[(\boldsymbol{W}, d\boldsymbol{W}), (\boldsymbol{b}, d\boldsymbol{b})]$$

And in the backward propagation, the gradient information is changed accordingly.

B. ReLU Module

The Rectified Linear Unit Activation Function (ReLU) is the most used activation function in the world right now, which can be found in almost all the convolution neural networks and other deep learning structures. Fig. 1 is the plot of ReLU function.

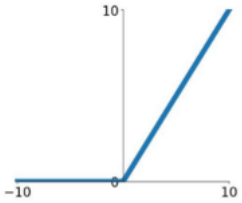

Fig. 1. ReLU funcition

1) Forward Propagation: The forward calculation of ReLU is defined as:

$$\boldsymbol{Y}_{ij} = ReLU(\boldsymbol{X}_{ij}) = \begin{cases} \boldsymbol{X}_{ij} & \boldsymbol{X}_{ij} > 0 \\ 0 & \text{otherwise} \end{cases}$$

2) Backward Propagation: As there is no parameters in ReLU layer, we only need to take care of the gradient with respect to the input, which can be calculated as following:

$$\left(\frac{d\boldsymbol{Y}}{d\boldsymbol{X}}\right)_{ij} = \begin{cases} 1 & \boldsymbol{X}_{ij} > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{d\boldsymbol{L}}{d\boldsymbol{X}} = \frac{d\boldsymbol{L}}{d\boldsymbol{Y}} \odot \frac{d\boldsymbol{Y}}{d\boldsymbol{X}}$$

where $\odot$ means element-wise multiplication.

The input of this layer is required in the gradient calculation, thus, need to be kept as a variable inside the module.

C. Tanh Module

Like logistic sigmoid function, the Tanh function is also sigmoidal, but instead outputs values that range (-1,1), as shown in Fig. 2. Thus strongly negative inputs to the Tanh will map to negative outputs. Additionally, only zero-valued inputs are mapped to near-zero outputs.

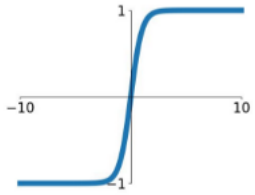

Fig. 2. Tanh funcition

1) Forward Propagation: The Tanh function is defined as:

$$\boldsymbol{Y}_{ij} = \tanh(\boldsymbol{X}_{ij}) = \frac{\exp(\boldsymbol{X}_{ij}) - \exp(-\boldsymbol{X}_{ij})}{\exp(\boldsymbol{X}_{ij}) + \exp(-\boldsymbol{X}_{ij})}$$

2) Backward Propagation: Based on the formulation above, the gradient w.r.t input is:

$$\left(\frac{d\boldsymbol{Y}}{d\boldsymbol{X}}\right)_{ij} = \frac{4}{(\exp(\boldsymbol{X}_{ij}) + \exp(-\boldsymbol{X}_{ij}))^2}$$

$$\frac{d\boldsymbol{L}}{d\boldsymbol{X}} = \frac{d\boldsymbol{L}}{d\boldsymbol{Y}} \odot \frac{d\boldsymbol{Y}}{d\boldsymbol{X}}$$

where $\odot$ means element-wise multiplication.

Like previous modules, the input of this layer is also required in the gradient calculation, thus, need to be kept as a variable inside the module.

D. Sequential Module

The sequential module work as a wrapper of different modules (Linear, ReLU, Tanh, etc.) to build a whole neural network. In this module, we need to consider the way to perform forward and backward operation for the sequence of modules and save the gradients in each layer.

In the module, we define a member function

add (self, *module)

which takes in several modules and store them as a list, which is a member variable, inside the Sequential module.

1) Forward Propagation: The forward pass is straightforward: iterate through the list of modules, and pass the output of one layer to the next layer as the input. Suppose there are k layers and the forward function is $\boldsymbol{f}_i$ for i^{th} layer, then we have

$$\boldsymbol{Y}_k = \boldsymbol{f}_k(\boldsymbol{Y}_{k-1}) = \cdots = \boldsymbol{f}_k(\cdots \boldsymbol{f}_1(\boldsymbol{X}))$$

2) Backward Propagation: Chain rule is the foundation of backward propagation, which provides us a technique for finding the derivative of composite functions. The backward pass requires the inverse iteration through the list of modules and passing the gradient of one layer to its former layer.

E. LossMSE Module

The Mean Squared Error (MSE) is a way of calculating the loss of neural network. It measures the average of the squares of the errors between the output of network $\boldsymbol{Y}$ and the ground truth $\boldsymbol{Y}^\star$.

1) Calculation:

$$\boldsymbol{L} = \frac{1}{N}\sum_{i=1}^{N}(\boldsymbol{Y}_i - \boldsymbol{Y}_i^\star)^2$$

2) Backward Propagation: The gradient of the loss w.r.t the output of network is the start point of a whole backward propagation. And it can be calculated as following:

$$\frac{d\boldsymbol{L}}{d\boldsymbol{Y}} = 2 \times (\boldsymbol{Y} - \boldsymbol{Y}^\star)$$

F. Mini-batch SGD

The stochastic gradient descend module in PyTorch is not as "stochastic" as it sounds. It actually needs to work with mini-batch and performs a mini-batch gradient descend. Thus, in our framework, we implement the mini-batch gradient descend instead of classical SGD.

II. Framework Testing

At last, we put our framework into practice and check the performance using a 2-class classification problem.

A. Data Generation

We generate a training and a test set of 1,000 points each sampled uniformly in $[0,1]^2$, with a label 0 if outside the disk of radius $1/\sqrt{2\pi}$ and 1 if inside. As shown in Fig. 3, blue points are with label 1 and right points are with label 0.

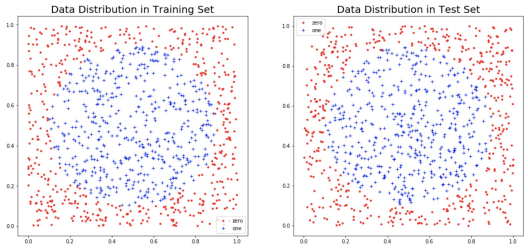

Fig. 3. Training and Test Data Distribution

B. Data Normalization

Data normalization is important in deep learning, which aims to have the same range of values for each of the inputs. This can guarantee stable convergence of weight and biases. Here, we normalize the data by subtracting the mean value of training data from training and test data, then divided them by standard deviation of training data.

C. Network building

The network used for testing has two input units, two output units and three hidden layers of 25 units. The activation function for each layer is ReLU, except for the last layer in which we use Tanh.

$$\text{Input} \rightarrow 3 \times \{\text{Linear} \xrightarrow{\text{ReLU}}\} \text{Linear} \xrightarrow{\text{Tanh}} \text{Output}$$

D. Training Configuration

In the training part, we need to first reshape the labels we generated above, a indicator vector, into a 2D matrix to match the size and also the range of value of the output of our network and suit for the calculation of MSE, as shown below:

$$\begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 1 \end{bmatrix} \Longrightarrow \begin{bmatrix} 1 & -1 \\ -1 & 1 \\ 1 & -1 \\ \vdots & \vdots \\ -1 & 1 \end{bmatrix}$$

The mini-batch gradient descend is applied to reduce the computation complexity. Parameters used in the settings is shown in the Table I.

TABLE I
PARAMETERS FOR TRAINING

Parameter	Value
Batch Size	50
Learning Rate	0.01
Epoch	10

E. Training and Test

Fig. 4 is the error log of 5 different run of training process. It's clear shown that the network and mini-batch gradient descend function well and the error rate is decreasing generally epoch by epoch. And the training error can reach as low as 5% after 10 epochs.

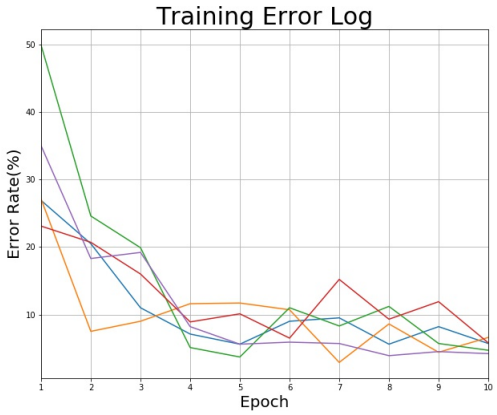

Fig. 4. Training Error Log

As for test error, although there is some difference between each run, the test error is around 5% for the most of the time.

III. Comparison with PyTorch

In this section, we use PyTorch to build a network with the same structure as the one from our framework. In the PyTorch network, **optim.SGD** and **nn.MSELoss** are used. As shown in Table. II, we test the two network models using the same dataset above and log the time and accuracy. Configuration are the same as shown in Table. I.

TABLE II
COMPARISON WITH PYTORCH (TIME AND ERROR RATE)

Our Framework			**PyTorch**		
Time/ms	Training/%	Test/%	Time/ms	Training/%	Test/%
222	5.5	5.5	260	8.3	10.3
249	11.9	10.6	252	1.4	5.6
231	7.0	5.4	257	4.7	6.5
217	4.9	4.1	245	7.4	12.7
222	6.4	5.3	255	3.7	9.0
221	7.7	7.4	264	2.7	6.4
234	15.5	13.9	301	6.0	7.9
226	5.6	4.8	255	5.9	8.6
216	5.4	4.8	258	5.7	11.2
235	4.0	5.0	259	3.6	8.5

As shown above, after 10 epoches with mini-batch gradient descend, both our framework and PyTorch can reach an error rate as low as 10 %. As for time, our framework runs a little bit faster than PyTorch because our framework is a just light version and doesn't use very complicate data structures or object definitions. Overall, the performance of two frameworks are similar.

IV. Conclusion

Through this project, we build a simple version of deep learning framework, including Linear Layer, ReLU Layer, Tanh Layer, Sequential, MSELoss and Mini-Batch Gradient Descend. In the implementation, many details need to be taken care of, such as tensor dimension, parameter update, etc. The framework we design is proven to function well on a simple 2-class classification problem and have similar performance with the PyTorch.