Mini Deep-learning Framework

FENG Wentao, SUN Zhaodong, WANG Yunbei

Abstract—**In this project, we designed a mini deep learning framework. The framework has modules including ReLU, Tanh, Linear, MSE loss and Sequential, SGD optimizer, and data loader. The basic structures of modules are also shown in the report. We also describe how we implement forward and backward propagation. During the training, we use the SGD optimizer to optimize our model and get losses and the error rate. Finally, we compare our framework with Pytorch.**

I. FRAMEWORK MODULES

1) Basic Module: The basic module is the base class which can be inherited by other modules such as ReLU, Tanh and Linear. The basic module defined some methods. If the output of the module is y and the input of the module is x. The backward method can calculate the gradient w.r.t the input given the gradient w.r.t the output. If the loss function is L, according to the chain rule, the general procedure is

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x} \tag{1}$$

If multiple modules are connected, the data can flow through backward and forward. The backward is also the implementation of the chain rule since we don't need to calculate the gradient of loss function w.r.t. the input and the gradient flow in the backward can do the chain rule. Besides, the param method can return the parameters such as weights and bias. The zero_grad method is to set all the gradient to be zero because when we do the backward, the gradient will be added to the last gradient. The __call__ method is to make our code more readable since we don't need to call forward method to do the forward calculation. The code of the basic module is defined below

```
class Module(object):

    def forward(self, *input):
        ...
    def backward(self, *gradwrtoutput):
        ...
    def param(self):
        ...
    def zero_grad(self):
        ...
    def __call__(self, *input):
        ...
```

A. ReLU activation

This ReLU module inherits from the basic module. We only need to override the forward and backward method. Since the module doesn't have any parameter, some methods don't need to be implemented. The forward function is

$$y = \max(x, 0) \tag{2}$$

The gradient of ReLU is

$$\frac{\partial y}{\partial x} = \begin{cases} 0 & x \le 0 \\ 1 & x > 0 \end{cases} \tag{3}$$

B. Tanh activation

This module has almost the same implementation as ReLU module. The forward function is

$$y = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{4}$$

The gradient of tanh is

$$\frac{\partial y}{\partial x} = 1 - \tanh^2(x) \tag{5}$$

C. Linear module

When the linear module is initialized, we have to specify the number of input dimensions, the number of output dimensions, and if we need the bias. Besides, the weights $\mathbf{W}$ and bias $\mathbf{b}$ are also initialized with uniform distribution.

$$\mathbf{W}, \mathbf{b} \sim U(-\frac{1}{\sqrt{N_{in}}}, \frac{1}{\sqrt{N_{in}}}) \tag{6}$$

Where N_{in} is the number of input dimensions. The initialization is consistent with the Pytorch Linear module, which uses the initialization in the paper [1].

Since the input data is in a batch, the input is the matrix $\mathbf{X}$ whose shape is (batchsize, input feature dimension). The weights $\mathbf{W}$'s shape is (input feature dimension, output feature dimension) and the bias $\mathbf{b}$'s shape is (1, output feature dimension). The forward is defined below

$$\mathbf{y} = \mathbf{X}\mathbf{W} + \mathbf{b} \tag{7}$$

The gradients are defined below

$$\frac{\partial L}{\partial \mathbf{W}} = \mathbf{X}^T \frac{\partial L}{\partial \mathbf{y}} \tag{8}$$

$$\frac{\partial L}{\partial \mathbf{b}} = \mathbf{1}^T \frac{\partial L}{\partial \mathbf{y}} \tag{9}$$

$$\frac{\partial L}{\partial \mathbf{X}} = \frac{\partial L}{\partial \mathbf{y}} \mathbf{W}^T \tag{10}$$

Besides, since the module has parameters and corresponding gradients, the param method must be implemented to return parameters and gradients. The zero_grad method also needs to be implemented to set all gradients zero.

D. Mean squared error(MSE) loss module

Mean squared loss is defined below

$$L = \frac{1}{N}\sum_{i,j}(\hat{y}_{i,j} - y_{i,j})^2 \tag{11}$$

where y is the ground truth data, $\hat{y}$ is the predicted output, i is the index of the data in a batch, j is the index of features and N is the size of y.

The backward method is a little different from the previous ones. Since the loss module is the last module of the model, it doesn't need to get the gradient w.r.t. the output $\partial L/\partial y$. The gradient of MSE loss is

$$\frac{\partial L}{\partial \hat{y}_{i,j}} = \frac{2}{N}(\hat{y}_{i,j} - y_{i,j}) \tag{12}$$

Since the loss module doesn't have parameters; the zero_grad and param method are not required.

E. Sequential module

The sequential module can contain several modules in a sequence. The forward and backward method use a loop to make the data flow through all layers. To add module to the sequential module, we only need to add them in the initialization, which is shown below.

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

This is similar to the Sequential module in Pytorch.

F. Optimizer module

We use stochastic gradient descent to optimize the model. Since this module is different from modules like Linear and activation functions, the module doesn't inherit from the basic module. We call the optimizer module SGD, which is defined below.

```
class SGD():

  def __init__(self, lr, model):
    ...
  def step(self):
    ...
```

When the module is initialized, the learning rate is required. Besides, the model is also required, which can be Sequential, Linear and so on. After running the backward to calculate gradients, we can call the step method to update weights. This module is also similar to torch.optim.SGD, but ours is a light version.

G. Dataloader module

I created an iterable data loader class. When the module is initialized, it will generate a dataset of 1000 points sampled uniformly in $[0, 1]^2$. If a point is in the circle with the radius $1/\sqrt{2\pi}$ and center $(0.5, 0.5)$, the corresponding label is 1 otherwise 0. However, the project requires the 2-dimensional output, so we transform the 1-dimensional label to 2-dimensional label. The label 1 is $(1, 1)$ while the label 0 is (-1, -1).

The data loader is also iterable, which can facilitate the data loading in the loop. The batch size has to be specified when the data loader is initialized. For each iteration, the module will return a batch of data, 1D label, and 2D label. The structure of the data loader class is defined below.

```
class dataloader():

    def __init__(self, batchsize):
        ...
    def __iter__(self):
        ...
    def __next__(self):
        ...
    def next(self):
        ...
    def __len__(self):
        ...
```

II. Implementation Details

A. Training data

As mentioned in the part of the Dataloader module, the dataset has the 2-dimensional input, 1D labels, and 2D labels. During the training, we will use 2D labels in the MSE loss. In testing, we will use 1D labels to calculate accuracy. The batch size is 10 in our experiment.

B. Neural network

We use fully connected layers with ReLU in the hidden layers. In the output, we use Tanh as the activation. Both the input layer and the output layer are 2-dimensional. There are three hidden layers with 25 neurons.

C. Training

The loss function is the mean squared loss. Before we run the backward pass, we have to set all gradients to zero. Then we can run the backward pass to calculate the gradients and use SGD to update parameters. The number of epochs is 50, and the learning rate is 0.1.

III. Results

A. Loss and Error

The model is trained for 10 rounds and we obtained the loss and errors. We calculated the mean of loss and errors on 10 rounds. In Figure 1, the red curve shows the mean of loss over 10 rounds and the gray curves show the losses of 10 rounds.

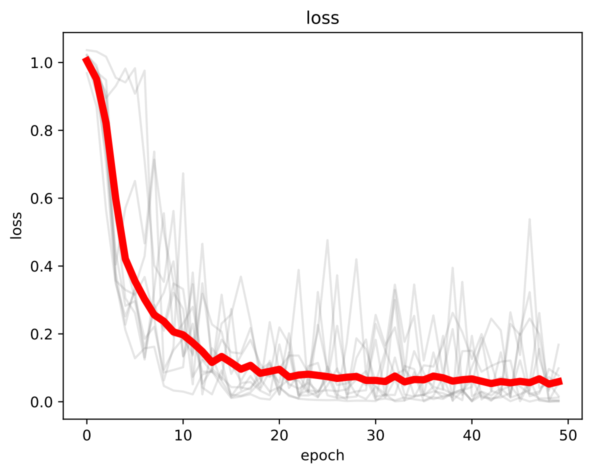

Figure 1: loss w.r.t epochs(average on 10 rounds)

In Figure 2, it shows the train and test errors. The train and test errors are very close, but the train error is slightly smaller than test error. The result is averaged over 10 rounds to make the curve smooth. After 50 epochs, the error can be about 4%.

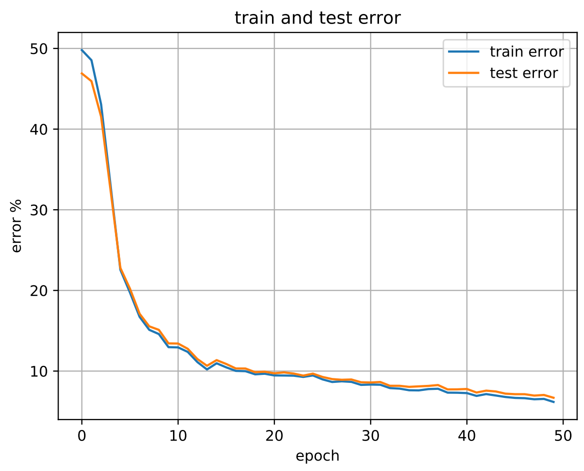

Figure 2: Train and test error w.r.t epochs(average on 10 rounds)

B. Comparison with Pytorch

We compare our framework with Pytorch. We use the same configuration as mentioned above. The number of epochs is 50, and the batch size is 10. The learning rate is 0.1. In Table I, the errors and running time for both frameworks are very similar, which verify that our Mini framework can work as well as Pytorch.

IV. Conclusion

We build some modules such as Linear, ReLU, Tanh, Sequential, MSE loss for the mini deep learning framework.

Framework	Train error	Test error	Time(s)
Ours	8.03%	8.73%	5.50
Pytorch	7.47%	8.81%	5.37

Table I: Comparison between our framework and Pytorch(10 rounds)

Besides, we also design extra modules such as optimizer and data loader to facilitate the training. To test if the mini-framework can work well, a simple model with fully connected layers is trained on our generated data. The performance of errors and running time is very similar to Pytorch.

References

[1] He, Kaiming, et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." Proceedings of the IEEE international conference on computer vision. 2015.