Implementation of a Mini Deep-Learning Framework

Andres Ivan Montero Cassab, Ali El Abridi
School of Computer Science and Communication Systems, EPFL

***Abstract*— The objective of this project is to implement from scratch a mini deep learning framework using only PyTorch's tensor operation and the standard math library, hence in particular without using autograd and neural-network modules available in PyTorch. Our framework consists of the forward and backward propagation implementation of several modules including Linear (fully-connected layer), Tanh, Relu, LeakyRelu, Sigmoid (activation functions), Sequential (combination of layers), and MSE (loss function) with mini-batch SGD to optimize the parameters. A 2-class classification is used to test the correctness, accuracy and performance of the framework.**

I. Framework Implementation

This section introduces the different modules in our framework including the mathematical principals underlying them, and how they are able to function together.

A. Module base class

The class *Module* serves as a base class for all neural network modules defined in the framework, i.e., Linear, Tanh, Relu, LeakyRelu, Sigmoid, Sequential, and MSE. It includes two abstract methods forward, and backward that raise an error if not implemented. All additional modules should inherit this class.

B. Linear Module

The linear module takes two mandatory input parameters, dimension of the input ***(in_features)***, dimension of the output (***out_features)***, and one optional Boolean input parameter representing the use of a bias in the current layer, or not *(Default: True)*, and keeps track of weights, bias, and the gradients.

1) Initialization: We initialize the weights $\boldsymbol{W}$ and biases $\boldsymbol{b}$ at each layer with the following commonly used heuristic:

$$W_{i,j} \sim \mathcal{U}(-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}), b_j \sim \mathcal{U}(-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}) \quad (1)$$

Where $\mathcal{U}[-a, a]$ is the uniform distribution in the interval $(-a, a)$ and n is the number of hidden units in the previous layer.

2) Forward Propagation: The linear module implements a fully-connected layer that applies a linear transformation to the upcoming input data $\boldsymbol{X}$.

$$Y = XW + b \quad (2)$$

3) Backward propagation: The gradients with respect to the input, and parameters (weights, and bias) are computed as follow:

$$\frac{dL}{dX} = \frac{dL}{dY}W^T, \quad \frac{dL}{db} = \frac{dL}{dY}, \quad \frac{dL}{dW} = X^T\frac{dL}{dY}$$

C. ReLU module

ReLU (rectified linear unit) is a type of activation function for neural networks that is the most commonly used for convolutional neural networks. ReLU is linear (identity) for all positive values, and zero for all negative. Figure 1 shows the plot of ReLU function.

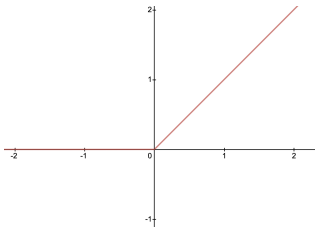

Fig. 1. ReLU function.

1) Forward Propagation: ReLU activation function implemented in the framework is mathematically defined as follow:

$$ReLU(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

2) Backward Propagation:

$$f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$

D. LeakyReLU module

LeakyReLU is a variant of ReLU activation function. It has a small slope for negative values instead of zero, and it has the benefit of fixing the *dying ReLU* problem and speeding up the learning process. Figure 2 shows the plot of LeakyReLU function.

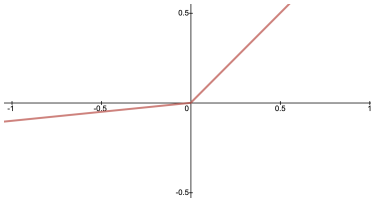

Fig. 2. LeakyReLU function.

1) Forward Propagation: LeakyReLU activation function implemented in the framework is mathematically defined as follow:

$$LeakyReLU(x) = \begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

2) Backward Propagation:

$$f'(x) = \begin{cases} 0.01 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$

E. Sigmoid module

Sigmoid activation function is one of the most widely used in classification problems. Unlike ReLU, the output of the activation is always in a bounded range (0,1). It performs well for classification by making clear distinction on predictions by bringing the activation to either sides of the curve. Figure 3 shows the plot of the Sigmoid function.

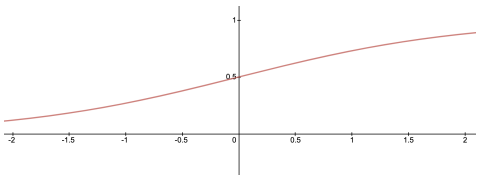

Fig. 3. Sigmoid function.

1) Forward Propagation: Sigmoid activation function implemented in the framework is mathematically defined as follow:

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

2) Backward Propagation:

$$f'(x) = f(x)(1 - f(x)) \tag{4}$$

F. Tanh module

Tanh activation function is a scaled Sigmoid. It is also a bound activation function in range (-1, 1) but with a steeper gradient. Figure 4 shows the plot of Tanh function.

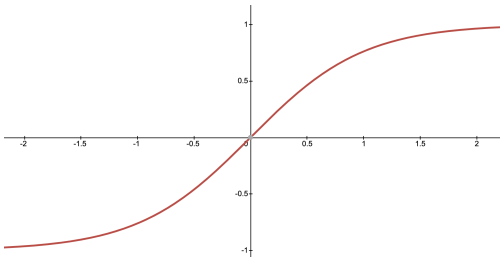

Fig. 4. Tanh function.

1) Forward Propagation: Tanh activation function implemented in the framework is mathematically defined as follow:

$$tanh(x) = 2 * sigmoid(2x) - 1 = \frac{2}{1 + e^{-2x}} - 1 \tag{5}$$

2) Backward propagation:

$$f'(x) = 1 - f(x)^2 \tag{6}$$

G. Sequential Module

The sequential module works as a container for different modules such as Linear, Tanh, Relu, etc. in order to build neural network structure. Added modules are stored in the order they have been passed in to the constructor in order to perform forward and backward operations for this sequence of modules, and update the layers' parameters. The sequential module constructor is defined as follow:

```
def __init__(self, layers)
```

which takes as an input a list of modules (layers), and initialize their weights, and store them.

1) Forward propagation: The forward propagation in the sequential module iterates through the list of layers and passes as input to a layer the output of the previous one. Suppose we have n layers defined in a sequential modules, f_i being the forward function of the i-th layer, and X being the input data, then the output of the n-th layer is defined as follow:

$$Y_n = f_n(Y_{n-1}) = ... = f_n(...f_1(X)) \tag{7}$$

2) Backward Propagation: The backward propagation in the sequential module implements the chain rule technique to find the derivative of a composite function by passing in reverse order of modules the output gradient of a layer to the previous layer.

H. MSE module

The Mean Square Error is used a loss function for the neural network to measure the average square error between the predicted value Y and the ground truth $\hat{Y}$.

$$L = \frac{1}{N}\sum_{i=1}^{N}(Y_i - \hat{Y}_i)^2 \tag{8}$$

The backward propagation uses the gradient of the loss w.r.t to the output of the neural network defined as follow:

$$\frac{dL}{dY} = 2 * (Y - \hat{Y}) \tag{9}$$

I. Mini-batch SGD

Due to the simplicity and lightness of the framework, the stochastic gradient descent optimizer implementation has been directly incorporated in the sequential modules (grad_step method), and linear module (update_param method).

II. Framework Testing

A 2-class classification is used to test the correctness, accuracy, and performance of the framework

A. Data Generation

To test the framework, we generated 1000 uniformly sampled points in $[0, 1]^2$ for the training and testing set. We label a point 1 if it is inside a disk of radius $\frac{1}{\sqrt{2\pi}}$, and 0 otherwise. The figure shows the distribution of the points with their labels.

1) one-shot encoding: The labels are then transformed into a one-shot encoding as the following.

$$\begin{bmatrix}1\\1\\0\\0\\.\\.\\1\\0\\1\\0\end{bmatrix} \Longrightarrow \begin{bmatrix}0 & 1\\0 & 1\\1 & 0\\1 & 0\\. & .\\. & .\\0 & 1\\1 & 0\\0 & 1\\1 & 0\end{bmatrix}$$

2) data normalization: In order to guarantee a stable convergence of the weights and biases in the training phase, we normalize the testing and training data by subtracting the mean and dividing by the standard deviation of the data.

B. Neural Network Structure

The neural network structure used to train the framework is composed of an input layer of two input units and output layer of two units, and in between three hidden layers with 25 units each, with ReLU as activation function for all except the last layer and Tanh as the activation function in the last layer.

C. Model Training and Testing

The training phase uses minibatch SGD shown in table I.

TABLE I
TRAINING PARAMETERS

Parameter	Value
Epochs	250
Learning Rate	0.05
Batch Size	100

Figure 5 shows the increase of the training accuracy set, and the decrease of the loss epoch by epoch in the training phase of the model.

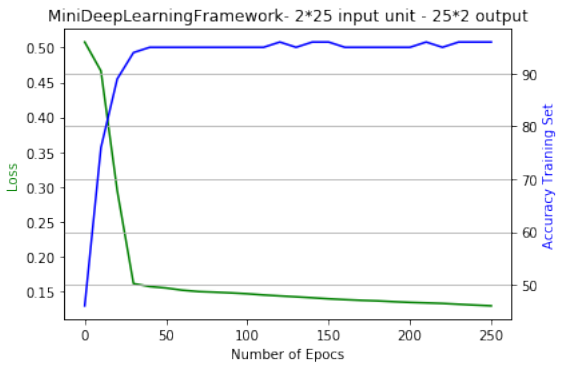

Fig. 5. Training accuracy vs Loss.

The training accuracy reaches as high as 98%, and the testing accuracy 97.2% in this example.

III. PYTORCH COMPARISON

In this section, the mini deep learning framework implementation is compared to the Pytorch framework in terms of accuracy and performance. The same neural network structure is created using optim.SGD, nn.MSELoss, and Autograd. Table II shows the comparison between the two models using the same parameter configuration used previously in table I.

TABLE II
ACCURACY AND PERFORMANCE COMPARISON

Mini Deep Learning Framework			**Pytorch**		
Training %	***Test%***	***Timing(ms)***	***Training%***	***Test%***	***Timing(ms)***
98.00%	97.20%	250	99.30%	98.20%	201
99.10%	98.30%	209	97.30%	96.40%	211
97.90%	96.00%	239	99.10%	97.00%	216
93.70%	91.20%	264	96.70%	95.00%	244
95.00%	94.40%	235	93.70%	91.00%	234
97.70%	96.00%	243	99.20%	98.10%	201
99.50%	98.5%	246	97.30%	96.20%	252
96.70%	96.00%	206	94.30%	93.00%	213
98.30%	97.20%	256	96.70%	95.60%	260
98.70%	97.00%	220	98.70%	96.30%	243
Mean					
97.00%	96.10%	236.06	97.60%	96.20%	226.57
std					
1.84%	2.10%	19.49	1.98%	2.23%	21.68

The accuracy and performance comparison between the mini deep learning framework and Pytorch framework show similar results. The mini deep learning framework and Pytorch both are able to reach a 96% accuracy on the test set with a standard deviation of 1.84%. In terms of performance, the Pytorch framework shows a slightly faster running time, and this might be due to the graph optimization implementation for the backward propagation while our deep learning implementation uses a naive implementation of it.

IV. CONCLUSION

The Mini Deep Learning Framework developed from scratch has the following modules: Module Base Class, Linear, ReLU, Leaky Relu, Sigmoid, Tanh, Sequential, and MSE loss. These modules implement forward and backward propagation. MSE function is used as a loss function along with SGD as optimizer to test our framework for a 2-class classification, and compare it with Pytorch framework. The resulting accuracy and performance are similar to Pytorch nn modules when compared with the same activation functions and neural network structure (FC-Relu, FC-Relu, FC-Relu, FC-Tanh). As we can observe in Table II, the framework developed is slightly slower than Pytorch by 10 ms, and the accuracy of 96% on the test set is similar to the accuracy of Pytorch.