Deep Learning Project1 Report

Chenkai Wang
323452
chenkai.wang@epfl.ch

Siran Li
321825
siran.li@epfl.ch

Shijian Xu
327448
shijian.xu@epfl.ch

***Abstract*—The objective of this project is to implement a mini "deep learning framework" using only PyTorch's tensor operations and the standard math library. The task is to use this mini framework to train a Multi-Layer Perceptron (MLP) for a binary classification problem. To achieve this goal, we implement the modules Linear, ReLU, Tanh, Sequential, MSELoss and SGD optimizer. The interface of these modules are very close to their PyTorch implementation.**

I. Implementation

The mini deep learning framework consists of several modules: `Linear`, `Sequential`, `MSELoss`, `SGD` optimizer and several activation functions, which are `ReLU`, `Tanh`, `Sigmoid` and `SELU` . All of these modules, except the `SGD` optimizer, inherit from a basic `Module` class. The final MLP model also inherits from the `Module` class and is composed of a `Sequential` member.

A. Modules

1) `Module`: The base class `Module` defines the functions each module needs to implement. The code snippet of the implementation of the basic `Module` class is shown below.

```python
class Module(object):
    """
    Base Class. Other modules should inherit from it
     and rewrite the ``forward'', ``backword'' and
     ``param'' functions.
    """
    def __call__(self, *input):
        return self.forward(*input)

    def forward(self, *input):
        raise NotImplementedError

    def backward(self, *gradwrtoutput):
        raise NotImplementedError

    def parameters(self):
        return []
```

Listing 1: The Base Module Class

We also use the Python built-in function `__call__` to make the modules' interfaces behave more PyTorch-like. In this case, we can call the module directly (`model(x)`) instead of `model.forward(x)`.

2) `Linear`: `Linear` layer is the core in the framework. In our implementation, this layer takes in a 2-d tensor $x \in \mathbb{R}^{N \times in\_features}$, where N is the batch size. The `forward` output is produced by $y = xW + b$, where W is the weight matrix with shape $in\_features \times out\_features$, b is the bias vector. Both W and b are initialized with uniform distribution from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{in\_features}$. The `backward` function takes in the gradient of loss with respect to the output y, $\frac{\partial l}{\partial y}$, and compute the gradient for W, b and x: $\frac{\partial l}{\partial W} = x^T \frac{\partial l}{\partial y}$, $\frac{\partial l}{\partial b} = \frac{\partial l}{\partial y}$, $\frac{\partial l}{\partial x} = \frac{\partial l}{\partial y} W^T$. The gradient of x will be returned and propagated to the former layer.

3) Activation Functions:

- `ReLU` The `ReLU` activation takes in the feature x and applies the following function element-wise:

$$y = ReLU(x) = (x)^+ = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \le 0 \end{cases} \tag{1}$$

  The gradient of `ReLU` is:

$$\frac{\partial y}{\partial x} = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \le 0 \end{cases} \tag{2}$$

- `Tanh` The `Tanh` activation takes in the feature x and applies the following function element-wise:

$$y = Tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} \tag{3}$$

  The gradient of `Tanh` is :

$$\frac{\partial y}{\partial x} = 1 - Tanh(x)^2 \tag{4}$$

- `Sigmoid` The `Sigmoid` activation takes in the feature x and applies the following function element-wise:

$$y = \sigma(x) = \frac{1}{1 + \exp(-x)} \tag{5}$$

  The gradient of `Sigmoid` is:

$$\frac{\partial y}{\partial x} = \sigma(x)(1 - \sigma(x)) \tag{6}$$

- `SELU` The `SELU` activation takes in the feature x and applies the following function element-wise:

$$y = SELU(x) = \text{scale} * \begin{cases} x & \text{if } x > 0 \\ \alpha e^x - \alpha & \text{if } x \le 0 \end{cases} \tag{7}$$

  where $\alpha = 1.6732632423543772848170429916717$, $scale = 1.0507009873554804934193349852946$. The gradient of `SELU` is:

$$\frac{\partial y}{\partial x} = \text{scale} * \begin{cases} 1 & \text{if } x > 0 \\ \alpha e^x & \text{if } x \le 0 \end{cases} \tag{8}$$

The final returned gradient is $\frac{\partial l}{\partial x} = \frac{\partial l}{\partial y}\frac{\partial y}{\partial x}$ and will be propagated to the former layer.

4) `Sequential`: A `Sequential` container takes in a list of layers as input and combines them together in order. In `forward` function, the layers' own `forward` functions are called sequentially to form a complete forward propagation. In `backward` function, the layers' own `backward` functions are called in reverse order sequentially to form a complete backward propagation.

5) `MSELoss`: `MSELoss` module computes the Mean Squared Error between the model output y and the target t:

$$\text{loss} = ||y - t||^2 \tag{9}$$

The final loss value is the mean value of the batch size samples. The returned gradient is $\frac{\partial l}{\partial y} = 2(y - t)$, averaged by the number of elements in the target tensor.

6) `SGD` Optimizer: The `SGD` optimizer updates the model parameters with the following rule:

$$x_{t+1} = x_t - \gamma \frac{\partial l}{\partial x_t} \tag{10}$$

where γ is the learning rate.

In our implementation, `SGD` optimizer provides two interfaces: `zero_grad` and `step`, which performs the same functions as the implementation in PyTorch. For loss backward propagation, there are some differences between our framework and PyTorch.

```
optimizer.zero_grad()    # zero the gradient buffers
output = model(input)
loss = criterion(output, target)
loss.backward()          # loss backward
optimizer.step()         # parameter update
```

Listing 2: PyTorch Loss Backward Propagation

```
optimizer.zero_grad()    # zero the gradient buffers
output = model(input)
loss = criterion(output, target)
dldy = criterion.backward()  # loss backward
model.backward(dldy)     # model backward
optimizer.step()         # parameter update
```

Listing 3: Our Framework Loss Backward Propagation

7) `MLP` model: The `MLP` model used for binary classification consists of 3 hidden layers, each with 25 unis. The input dimension is 2 while the output dimension is 1. Using the above implemented modules, the model definition in our framework is the same as in PyTorch, using a `Sequential` container to wrap the `Linear` and activation layers.

II. Experiment Results

To test our framework, we generate a training and test set of 1000 points sampled uniformly in $[0, 1]^2$. To each of them, we associate the label 0 if the point is outside the disk of radius $\frac{1}{\sqrt{2\pi}}$, and 1 otherwise. Therefore, the objective of the model is to predict if a given point is outside or inside the disk, a binary classification problem.

A. Effectiveness of Adding `Sigmoid`

We implement 4 activation functions: `ReLU`, `Tanh`, `SELU` and `Sigmoid`. It is strange to see that the model with all sigmoid activation can not be trained. Why ??? For binary classification tasks, usually a `Sigmoid` activation functions will be applied to the output. We verified the necessity of adding `Sigmoid` at the end by experiments. Each models are trained for 300 epochs, with batch size 100 and learning rate 0.1. The training losses are shown in Fig. 1.

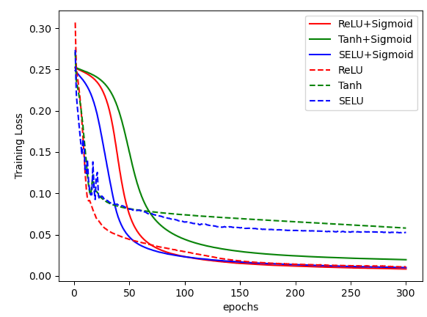

Fig. 1: The training loss of different activation (combinations).

From this figure, we can see that adding `Sigmoid` activation function at the end of the model smooths the training loss curves and results in lower training losses.

B. Comparison of Different Activation Functions

From Fig. 1 we can also see that models with `ReLU` and `SELU` have lower training losses than the model with `Tanh`. This can also be verified by the final test accuracy, as shown in Fig. 2. We train and test the models with `ReLU`, `Tanh`, `SELU` for 10 rounds, with batch size 100 and learning rate 0.1.

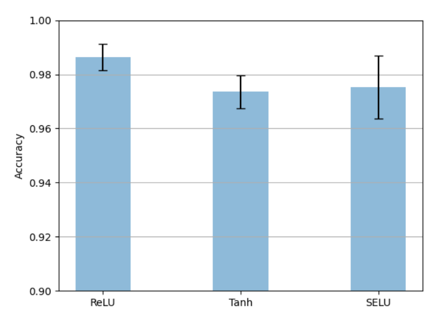

Fig. 2: The training loss of different activation (combinations).

III. Summary

There are some limitations in our framework. For binary classification tasks, `MSELoss` is not a good choice. It is better to use the binary cross entropy loss.