EE-559 Deep Learning Mini-Project 2 Report

Parth Kothari & Anshul Gupta

May 2021

1 Introduction

In this project, we build a mini-deep learning framework to train simple feedforward architectures. The framework is built using the standard tensor operations in PyTorch [1]. We demonstrate its capability by training a 3 layer feedforward network for a classification task. The network achieves a test error of $\sim 3\%$. We also show that the errors are very close to the same model trained with PyTorch.

2 Modules

Our framework supports the basic modules necessary to build a network. The general structure of a module is as follows:

```
class Module(object):

        # returns an operation on the input
        def forward(self, *input):
                raise NotImplementedError

        # accumulates the gradients w.r.t. the parameters
        # returns the gradient w.r.t the input
        def backward(self, *gradwrtoutput):
                raise NotImplementedError

        # returns the parameters and gradients w.r.t. the parameters
        def param(self):
                return []
```

Additionally, the Linear, ReLU [2], Tanh and Sequential modules support the `zero_grad` operation to clear the accumulated gradients w.r.t the parameters after each batch of samples.

2.1 Linear Module

The linear module returns a linear transformation on the input. It is initialized by specifying the input and output dimensions. An example usage is given below.

```
layer = Linear(2, 10)
inp = torch.empty(2).uniform_(0, 1)
out = layer.forward(inp)
```

2.2 ReLU and Tanh Modules

The ReLU and Tanh modules apply a non-linear transformation on the input. An example usage is given below.

```
activation = ReLU()
inp = torch.empty(2).uniform_(0, 1)
out = activation.forward(inp)
```

2.3 Sequential Module

The Sequential module allows the combination of the Linear, ReLU and Tanh modules to build a feedforward network. It is initialized by giving a list of the modules to combine. An example usage is given below.

```
model = Sequential([
                    Linear(2, 25),
                    ReLU(),
                    Linear(25, 1),
                    Tanh()
                    ])
inp = torch.empty(2).uniform_(0, 1)
out = model.forward(inp)
```

2.4 LossMSE Module

The LossMSE module computes the mean squared error loss between the prediction and the target. An example usage is given below.

```
mse = LossMSE()
pred = torch.empty(1).fill_(0.8)
target = torch.empty(1).fill_(1)
loss = mse.forward((pred, target))
```

3 Extension - Mini Batch SGD

Our framework supports mini-batch stochastic gradient descent for optimizing the parameters of the network. For each batch of samples the gradients with respect to the parameters are accumulated. The optimizer then performs gradient descent to update the parameters of the network. The next step is to clear the accumulated gradients. This can be done by calling the `zero_grad` function on the model. An example usage is given below.

```
for nb in range(num_batches):
    for i in range(batch_size):
        op = model.forward(X_train[idx])
        loss = mse.forward((op, Y_train[idx]))
        model.backward(mse.backward())
        idx += 1

    # update model params for each batch
    SGD(model.param(), alpha=0.001)
    model.zero_grad()
```

4 Experiments

We test our framework by training a feedforward network for a binary classification task. The network consists of an input layer with 2 units, 3 hidden layers with 25 units, and an output layer with 1 unit. Synthetic data is generated by sampling points from $[0,1]^2$. Points inside the disk centered at $(0.5, 0.5)$ and radius $1/2\pi$ have label 1. Points outside the circle have label 0. The network is trained using MSE loss for 100 epochs. We use the mini-batch SGD (Section 3) optimizer with a learning rate of 0.001.

4.1 Results

We report the train and the test errors of the model for different batch sizes in Table 1. Results are computed for 10 runs with random initializations for the network parameters and data. We report the means and standard deviations of the errors. We note that a batch size of 16 gives the least train and test errors.

Table 1: Train and test errors of the model on synthetic data. We compare performance for different batch sizes.

Batch Size	4	8	16	32	64	128
Train Error (%)	2.79 ± 1.12	2.86 ± 1.11	2.57 ± 0.79	3.59 ± 1.60	4.86 ± 2.61	10.07 ± 5.23
Test Error (%)	3.24 ± 0.92	3.08 ± 0.98	3.02 ± 1.14	3.85 ± 1.55	5.52 ± 2.61	10.68 ± 5.01

Table 2: Train and test errors of the model on synthetic data. We compare performance for the model trained using our framework and the model trained using PyTorch (batch size=16).

	Train Error (%)	Test Error (%)
Our framework	2.57 ± 0.79	3.02 ± 1.14
PyTorch	2.55 ± 0.81	3.05 ± 1.13

We also compare results for the model trained using our framework, and the model trained using PyTorch in Table 2. Once more, we report the means and standard deviations of the errors over 10 runs. It is of note that the errors for the model trained using our framework, and the model trained using PyTorch are very close to each other. We have verified this phenomenon for different random seeds. This further demonstrates the capability of our framework.

References

[1] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[2] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *International Conference on Machine Learning*, 2010.