EE-559 Deep Learning Mini-Project 2

Haeeun Kim, Valentin Loftsson, and Antoine Madrona

May 2021

Introduction

We present `lamp` 🪔, a lightweight deep learning framework written in Python using PyTorch. The framework was tested for binary classification on synthetic data using SGD. In this report, we discuss the architecture of the framework, its features and implementation details, advantages and disadvantages, and test performance.

Architecture

The architecture of the framework is illustrated in Figure 1. It is split into three main components: data loaders, modules, and optimizers. It was designed with extensibility in mind to facilitate the expansion of the framework.

Implementation and Features

The framework depends only on PyTorch's tensor operations and the standard math library. It was implemented such that the syntax resembles PyTorch syntax.

All neural network modules inherit from the `Module` class and each module needs to implement at least the `forward()` and `backward()` methods. The `Module` class also provides the `store_inputs()` method that subclasses need to call in `forward()` to enable the computation of the gradient in `backward()`. By default, the `parameters()` method returns an empty list. Currently, the only module in the framework that has parameters is the `Linear` module. The `Parameter` class is used internally for convenience to represent a parameter and its gradient.

The `Linear` module implements a fully-connected layer of arbitrary dimensions. It provides two types on parameter initialization: (1) the default initialization from the normal distribution $\mathcal{N}(0, 1)$ (2) Xavier initialization from $\mathcal{N}(0, \sigma^2)$ where

$$\sigma = \text{gain} \cdot \sqrt{\frac{2}{N_{l-1} + N_l}}$$

for layer l. This type of initialization avoids vanishing gradients during the forward pass. The gain is a scaling parameter that depends on the activation function used.

The `Sequential` class provides a container for modules. This facilitates building neural networks that consist of multiple layers and activations. Modules will be added to it in the order they are passed in the constructor.

Optimizers inherit from the `Optimizer` superclass which provides the common `zero_grad()` method to subclasses. All optimizers need to implement a `step()` method, that updates the parameters. This method needs to be called after gradients have been computed with `backward()`.

The `DataLoader` class facilitates iterating over minibatches of data samples. Users can enable shuffling the data before each epoch through the `shuffle` parameter. This shuffles the data internally before each epoch.

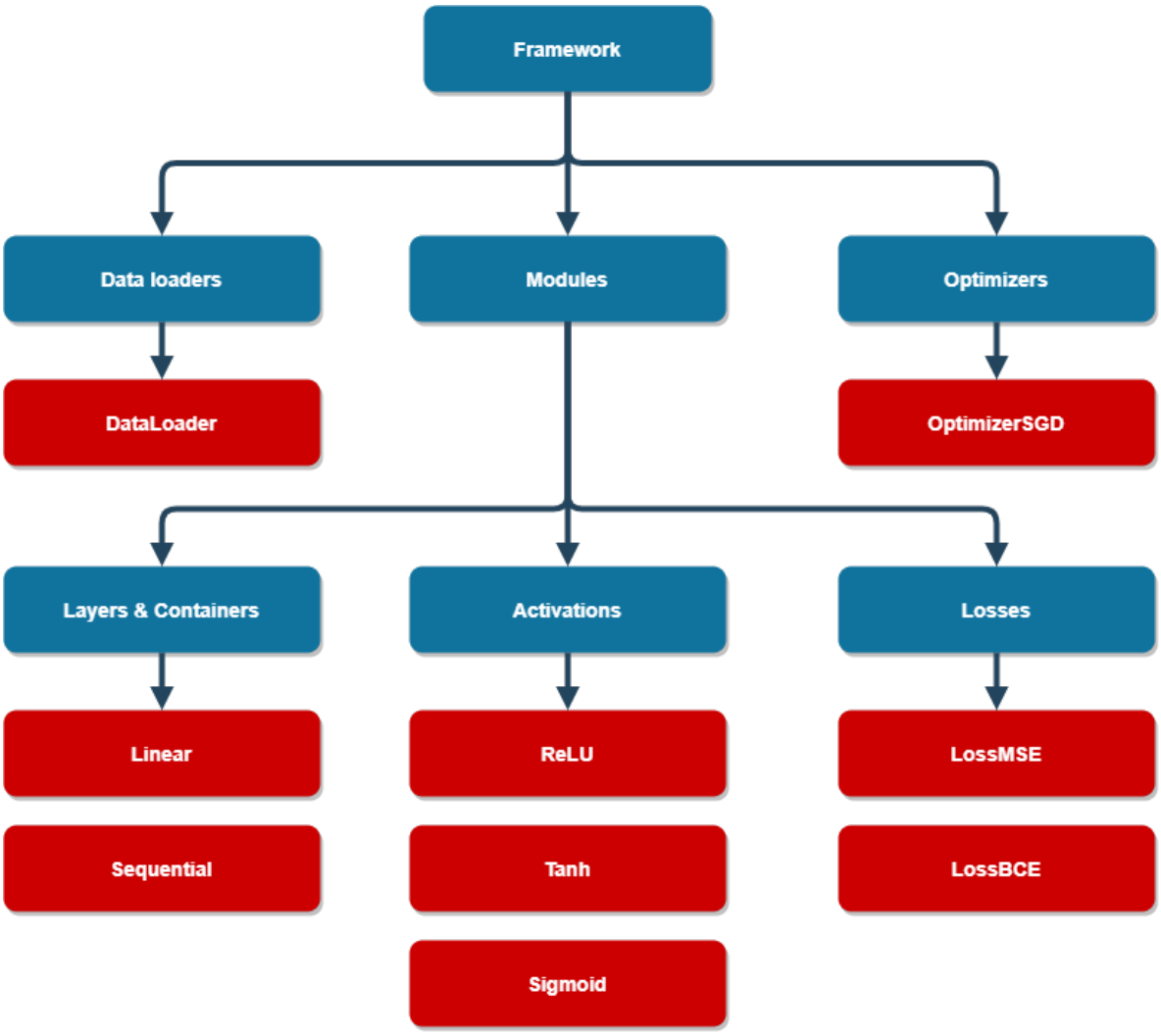


Figure 1: The framework architecture. Blue boxes represent categories and red boxes represent classes

Usage

The entire framework can be imported from `lamp.py` using `import lamp`. Example of usage can be found in `test.py` and `helpers.py`.

Advantages and Disadvantages

The `lamp` framework shares some features with PyTorch. However, it lacks many commonly used modules, functionalities, and parameters. For instance, it does not provide a convolutional layer module. Also, it cannot be used efficiently to train on very large datasets since computation cannot be moved to the GPU and no particular optimization measures were taken in its implementation.

Framework Testing and Conclusion

We provide a test file `test.py` that tests the framework on a synthetic dataset (see Figure 2). The network and the objective function used are as per the project outline. The activation function for each hidden layer is Tanh. We use the Sigmoid activation at the end to get an output in $[0, 1]$ which, in turn, is rounded to obtain the predicted class.

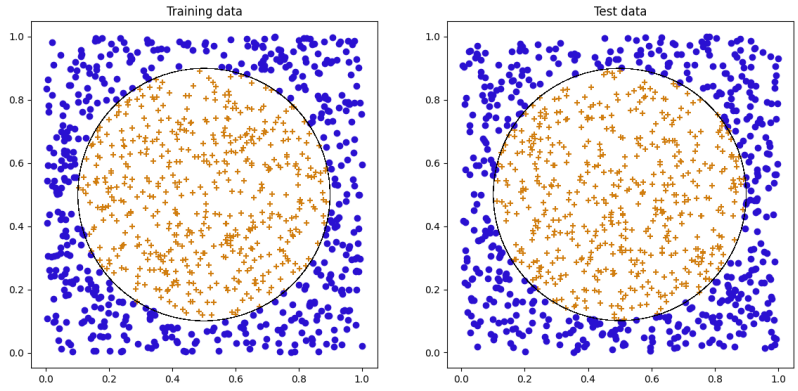
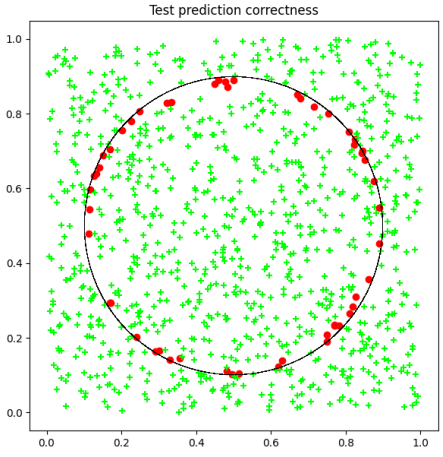


Figure 2: Training and test data sets. Each includes 1,000 points sampled uniformly from $[0, 1]^2$*. Each point is associated with a binary label, 0 if it is inside the disc centered at* $(0.5, 0.5)$ *of radius* $1/\sqrt{2\pi}$ *and 1 inside.*



(a) Example of test prediction correctness of a trained model. Red dots represent incorrectly classified samples and green pluses correctly classified samples.

(b) Test accuracy over 50 cycles

Figures 3a and 3b illustrate the test results. We report an average of 96% test classification accuracy over 50 cycles of training, testing, and evaluating—regardless if shuffling was performed or not.

To conclude, we have built a deep learning framework that can be used for typical classification scenarios. It performs very well on the given test scenario. The design of the framework allows for its future expansion.