Deep Learning Project

Dimitri Coukos, Paola Malsot, and Niels Mortenson

Mai 2019

1 Introduction

In this mini-project we aimed to implement a mini-framework mimicking PyTorch. This mini-framework should have similar usage to PyTorch so that it can be familiar to the user and easy to use. Additionally it should be coded in a straightforward manner so that it can be easily extended. Furthermore, it should efficiently and accurately classify the dummy data we give to it.

The structure of the frame-work is implemented with a single superclass *Module* which all other implemented classes inherit. This structure allows all implemented classes to have a familiar structure and usage.

Multiple steps were taken during the coding of this project to ensure that the implemented framework would be efficient. Importantly, we frequently used matrix operations to ensure that iteration through the data would be minimal. Additionally, training is optimized through a simple algorithm which adapts the learning rate as the training process continues. In order to minimize the amount of data which is passed between layers, activation layers are implemented as attributes of the layers.

In order to demonstrate the efficacy of this frame work we implement a simple network which uses only linear layers. This network has two input neurons, two output neurons, and three hidden layers with 25 neurons each. The framework, of course, is functional with other architectures. Two types of activation layers are implemented: rectified linear unit (ReLU) for the hidden layers, and hyperbolic tangent (Tanh) for the output layer.

2 Design and Implementation

In order to construct a mini-framework mirroring PyTorch, we started by implementing a number of modules, with the goal of making the code easy to reuse, and easy to understand for users already familiar with PyTorch. *Module* is a superclass which defines an outline for the forward, backwards, and update functions which are used during classification and training of the network. Defining these in the superclass ensures that inheriting modules respect the overall structure of the framework and are able to participate in the training of the network. These inheriting modules represent the basic structure of the deep network (*Sequential*), individual layers (*Linear*), and activation layers(*ReLU* and *Tanh*). We chose to implement the activation layers as an attribute of each instance of a layer, as we found that this

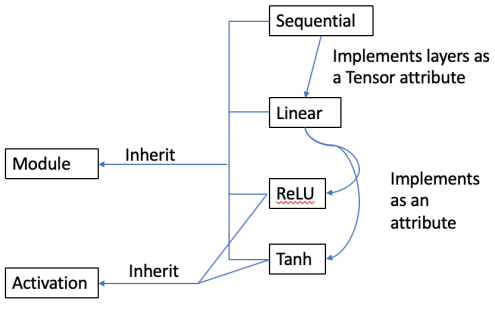

Figure 1: Schematic of the class hierarchy within the Modules implemented in our mini-framework.

simplified greatly the structure of our network-level functions, and required that we pass less information back to the Sequential modules while running backpropagation. As each layer had an activation function regardless, we found that this implementation was both straightforward and intuitive. In order to accommodate the classification needs of our network, we implemented Tanh, which is used for the output layer, and ReLU which is used for the hidden layers. Furthermore, the framework asserts that the defined activation function is a valid activation function, which consists simply in checking wither the activation function is an instance of *Activation* which is an empty superclass. The applicability of our framework to other, larger classification problems is ensured by our use of Xavier Initialization, which prevents the gradient from vanishing or exploding in very deep architectures.

The framework's ability to train a network is based on three functions which are defined in the *Module* superclass: forward, backwards, and update. These are built without using PyTorch's autograd functionality. During classification, forward is first called in Sequential, where it then calls the forward function in each of its modules, so that the weighted sums from each layer's neurons are propagated forward through the network. In order to update the network, it is necessary to effectuate backpropagation, and for this the backwards functions are called from each of the modules in *Sequential* in reverse order. The derivative of the loss with respect to the outputs of the architecture is calculated using a function which implements a gradient of our loss function, the mean squared error. One step of gradient descent is performed in the update function. Like forward, and backwards, calling *Sequential*'s update function will lead to all the sub-modules' update functions being called as well. With a default batch size of 100 samples, our framework implements stochastic mini-batch gradient descent, but this can be changed to stochastic or batch gradient descent by changing the size of the batch.

In order to optimize the efficiency of the network we took a number of steps which allow for the optimization of the learning rate, the minimization of iteration through the data, and the minimization of passing data between modules. Firstly, in order to minimize the number of times data would need to be passed between modules, each *Linear* modules calls its activation function module directly. This avoids needing to pass and store data in the higher level *Sequential* module. Additionally, we structured our framework so as to maximize the amount of matrix-level operations. Looking through the forward and backwards functions, the reader might note extensive use of reshaping and the *torch.matmul* function, which allow us to avoid iterating through the samples, and instead process entire batches in one step in the code. Finally, during the training process the learning rate is updated automatically using a simple algorithm, which allows it to be robust to different batch sizes, and to automatically choose an optimal learning rate. This algorithm compares the current training error to the training error in the two previous epochs. If the training error has increased with respect to the previous epoch, the learning rate is decreased by a factor of 0.999. If the training rate has also increased with respect to the before-previous epoch, the learning rate is decreased additionally by a factor of 0.998.

A number of functions were implemented with the goal that they should be reusable. As such, they are not incorporated in modules, but rather in a separate file, *functions.py*. These functions include mathematical tensor operations such as ReLU, the calculation of the loss, the generation of dummy data, and a variety of derivatives of useful functions, which are necessary for backpropagation.

To make the structure of the framework more intuitive we separated the important parts of the framework over 4 files.

- test.py: This file implements the network, trains it, and tests it.
- modules.py: This file implements the classes representing the network, its layers, activation functions. In addition, this file is where the forward, backward, and update functions are defined.
- functions.py: This file defines the data generation functions, the mathematical functions that the activation functions are based on, and their derivatives.
- config.py: This file contains parameters which define the debugging status, as well as parameters which define the batch size and the number of epochs for training.

3 Testing & Performance

We tested our model with an initial learning rate of 0.2, a batch size of 100 samples, and 2000 epochs. Although a lower number of epochs gave us a good performance, we wanted to observe whether the architecture could eventually predict the testing set perfectly. Perfect prediction accuracy was also obtained using batch-sizes of 1, 10 and 1000. Perfect prediction accuracy was a reasonable expectation, given that the test dataset's classes were attributed deterministically based on the sample features. With the above conditions, we did indeed reach perfect prediction accuracy. Interestingly, we did not reach perfect training accuracy, likely because rounding errors during the data generation may have led to misclassifications.

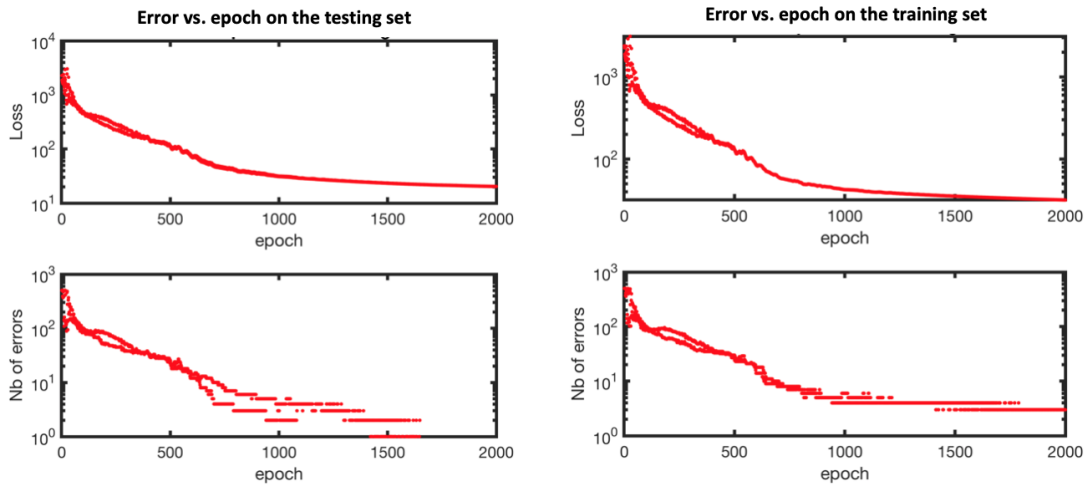

Figure 2: Training (Right) and Testing(Left) loss and errors. After about 1700 epochs, our architecture reaches a prediction accuracy of 100with our dummy data.