EE-559
Deep Learning
Spring 2021

Project 2
Mini Deep-Learning Framework

Authors
Jon Kqiku
Mert Soydinc
Alejandro Noguerón

Professor
François Fleuret

November 20, 2021

1 Introduction

The goal of this project is to build a hand-made simple neural network framework only using PyTorch tensor operations and python's math library, that is, only importing `torch.empty` and `math`. The framework must allow to build build sequences of fully connected layers with Tanh and ReLU as activation functions. Of course, it must implement the forward and backward pass of the architecture to train it on data optimized with stochastic gradient descent on the mean squared error loss.

Our framework must then be tested on a toy example. The toy data set consists of two-dimensional inputs in $[0, 1]^2$ and one-dimensional targets in $\{0, 1\}$. The target is 1 if the corresponding input lies inside the disk centered $(0.5, 0.5)$ with radius $1/\sqrt{2\pi}$ and 0 otherwise.

In the next section, we give details on our implementation of this framework. Then we discuss the results of a simple architecture built with our framework and trained on the toy example. We also compare these results with the results a pytorch implementation of the same architecture yields. We end this report with a conclusion...

2 Methods

Our framework enables us to build and train a neural network with fully connected layers optimized for the MSE loss with SGD. On top of the required Tanh and ReLU activation functions, our framework also supports the Sigmoid function and can incorporate Drop-Out and Batch-Normalization.

Supported network components	
Parametrized	Non-parametrized
Fully connected layer	Tanh
Batch-normalization	ReLU
	Sigmoid
	Drop-out
	MSE

Table 1: Components implemented in our framework

For the implementation, we consider each operation (or component) to be a layer in the network, so that we can build it in a sequential manner with the `Sequential` class, with an ordered list of string named layers as argument, among other arguments. Each layer is designed as a class of its own, all featuring a `forward` and `backward` method where the former returns the output after the component's operation and the latter computes the gradient with respect to that component. In addition, parametrized components have a `update` module to update their parameters during the backward pass. Moreover, arguments must be passed to initialize their respective constructors, such as the input and output size of the fully connected layer. Learnable parameters for Fully connected layer and the Batch-normalization are initialized randomly according to the initialization strategy used by Pytorch for its respective modules[2, 1].

The network construction as well as the network training are both implemented in the above mentioned `Sequential` class. An instance of the network is built by passing a list of strings representing the names of the components in the order in which they appear in the architecture, and a list

of lists representing the arguments of the components, which are empty lists for non-parametrized components with the exception of the Dropout component which takes the probability of the drop as its parameter. `Sequential` essentially creates a doubly link list of layers(layers are linked by their forwards and their backwards functions) and manages the creation of the layers for the users convenience.

The `train` method implements a single forward and backward pass. It uses the attribute `layers` to sequentially compute the output of a layer until it reaches the penultimate one, point at which it computes the loss. The backward pass is implemented by calling the `backward` method to the penultimate element of `layers`, which is the output of the model, and an instance of a component's class. Since every component class has an attribute `prev_module` initialized with the instance of the previous layer and since every `backward` method also calls it on `prev_module`, the backward pass will reach the first layer with only one explicit call. Parameters of parametrized components are updated when `backward` is called for that component which also calls `update`. Parameter updates are performed by calculating the gradient w.r.t. the parameter. Then we adjust the parameter according to the gradient, the given learning rate and the given momentum value.

We also implemented two helper classes called the `ModelTrainer` and `ModelTester` which train or test the given model according to their parameters. The user just needs to provide the hyper-parameters such as a model, train data, train targets, number of epochs, learning rate or momentum to these classes.

3 Results & Discussion

For the toy example, the model we consider has two input units, two output units, three hidden layers with 25 units each, Tanh activation at the first hidden layer, ReLU activation in the second hidden layer and Tanh again at the last hidden layer without activation in the final output layer. We train this model with a learning rate of 7e-2, momentum rate of 0.15 using 1000 epochs with a batch size of 20. Since parameters are intialized randomly, performance can vary from initialization to initialization. We thus train this model 100 times, reinitializing the model at each round.

	Train MSE		Test accuracy	
	Our framework	PyTorch	Our framework	PyTorch
Mean	0.057	0.058	0.9605	0.9517
Std	0.006	0.007	0.0112	0.0118
CV	0.112	0.119	0.0116	0.0124

Table 2: Comparison of performance statistics

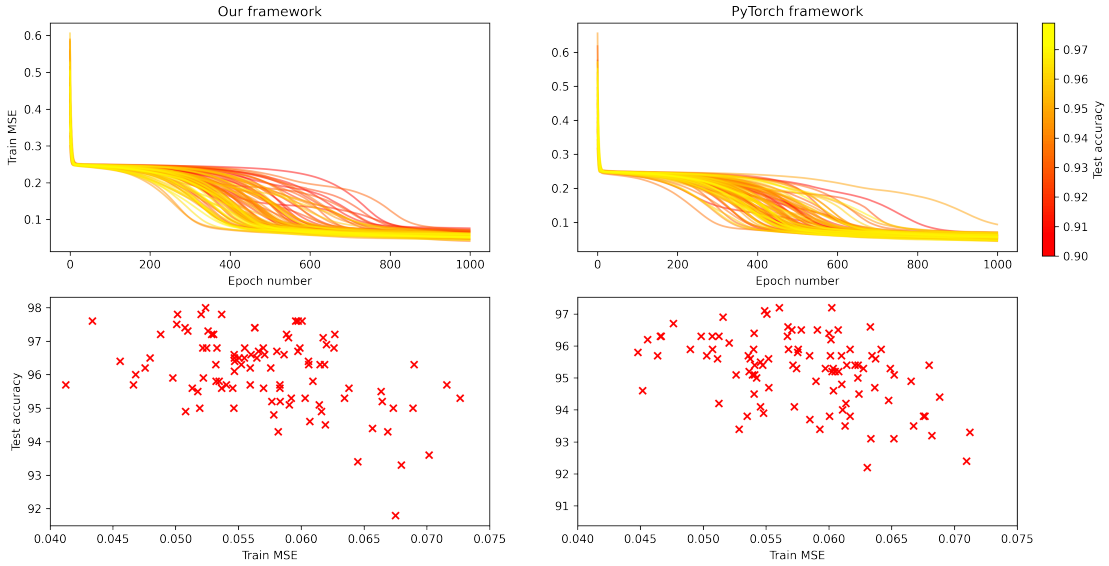

Figure 1: Training loss and test accuracy through 100 rounds for both implementations

Our framework achieves a slightly lower mean training loss (in the last epoch) than PyTorch's implementation, as well as a slightly higher mean test accuracy. Moreover, variance of both metrics is lower with our implementation, also resulting in lower coefficients of variation, implying a slightly more stable training with respect to initialization, although these differences might be insignificant.

4 Conclusion

In the end, the model created by our architecture behaves very similarly to the model created by Pytorch. We see that the loss decrease with respect to the number of epochs follows the same pattern in both of the models. We also see the very similarly shaped clusters forming in the accuracy vs MSE graphs. Our models also achieves upwards of 95 accuracy in the toy example for the test set. Considering that Pytorch is a very popular and thoroughly tested library, the above results strongly suggest that our architecture is able to accurately train and test models using the components we implemented.

References

[1] Pytorch batch normalization module. `https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm1d.html`. Accessed: 2021-05-24.

[2] Pytorch linear layer module. `https://pytorch.org/docs/stable/generated/torch.nn.Linear.html`. Accessed: 2021-05-24.