Mini-Project 2: Deep Learning Framework

Adrian Guerra Ayyoub El Amrani Jan Benzing
EE-559 Deep Learning 2019 EPFL

***Abstract*—The objective of this project is to build and design a miniaturized deep learning framework, that provides the necessary tools to build networks combining fully connected layers, hyperbolic tangent (TanH), and Rectified-Layer Unit (ReLU), run the forward and backward passes and optimize parameters with stochastic gradient descent (SGD) for mean-squared error (MSE). Everything is coded from scratch using no pre-existing neural-network python toolbox and using only the basic PyTorch operations. This project is part of EE-559 Deep Learning course taught at EPFL during the spring semester 2019.**

I. Introduction

Today, we have a myriad of frameworks at our disposal that allows us to develop tools that can offer a better level of abstraction along with simplification of difficult programming challenges.

Some popular deep learning frameworks include TensorFlow, PyTorch or Keras. In *EE-559 Deep Learning* we focus on the PyTorch framework, primarily developed by Facebook's artificial-intelligence research group [1]. PyTorch provides libraries for basic tensor manipulation on CPUs and GPUs, a built-in neural network library, model training utilities, and a multiprocessing library that can work with shared memory [2]. In particular, it uses an automatic differentiation technique called autograd where a graph that records the operations performed replays it backwards to compute the gradients.

The objective of this project is to design a miniaturized deep learning framework using only PyTorchs tensor operations and the standard `math` library. In particular, without using autograd or the neural-network modules.

Our framework provides the necessary tools to build networks combining fully connected layers, hyperbolic tangent (TanH), and Rectified-Layer Unit (ReLU), run the forward and backward passes and optimize parameters with stochastic gradient descent (SGD) for mean-squared error (MSE).

We explore the design choices of our framework, how to implement a model, our results using the architecture and data set instructed, discuss our findings and discuss further work.

II. Design

A. Modules

Similar to PyTorch, we define classes `Sequential`, `Linear`, `ReLu`, `TanH` and `MSELoss` to inherit from a base abstract class `Module`, as shown in Figure 1.

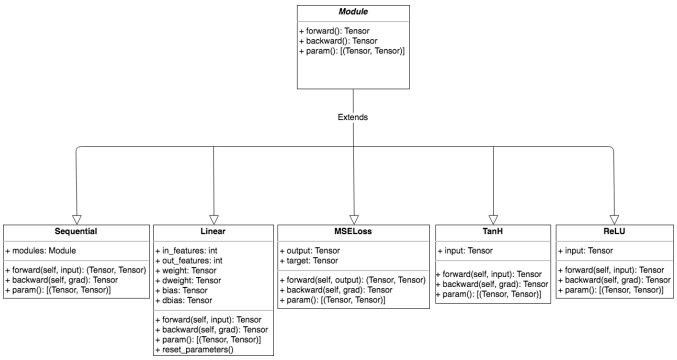

Fig. 1: UML Class Diagram of modules

`Module` provides the basic methods all modules have, namely `forward`, `backward` and `param` shown in Listing 1.

Listing 1: Module class

```
class Module(object):
    def forward(self, input):
        raise NotImplementedError
    def backward(self, gradswrtoutput):
        raise NotImplementedError
    def param(self):
        return []
```

B. Optimization

As mentioned, SGD is the optimization technique used in this project with the MSE as a loss metric. In this light `optimizer.py` contains an `SGD` class initialized with the learning rate and the parameters to optimize. As shown in Figure 2, two methods are defined: `step()` and `zero_grad()`, which update the weights and bias in every epoch and reset the gradients to zero respectively. The name between the file and the class differ, in case we were to add more optimizers (see Section IV).

C. Additional methods

Additionally, we define python files with the methods shown in Figure 2.

`init.py` contains the functions necessary to perform the initialization of modules. In our case, we use it to initialize the weight tensors using a Xavier initializtion [3].

`functions.py` is analogous to `Functional` in PyTorch and contains the mathematical functions necessary for the training of our model.

Finally, `helpers.py` contains methods for the generation of our data set, namely a function that generates a training

and a test set sampled uniformly in $[0,1]^2$ with a label 0 if outside the disk of radius $1/\sqrt{2\pi}$ and 1 inside.

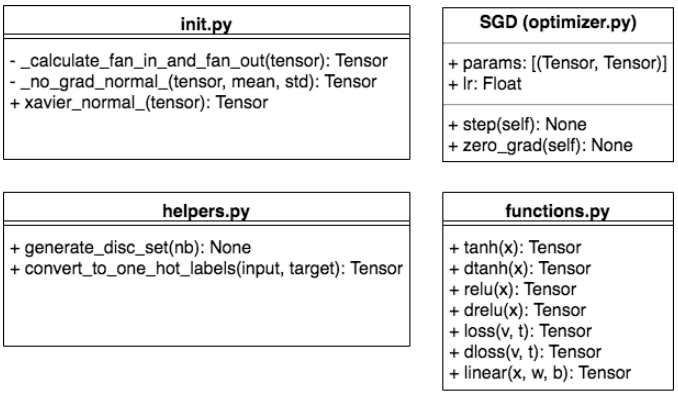

Fig. 2: Additional python files and SGD class

D. Implementation

`Sequential` is the container that enables to build a model. Modules are added to it in the order they are passed in the constructor. Forward and backward passes are performed by simply calling `forward` and `backward` respectively on the model, thereby making our framework very easy to use. Our implementation throws an error whenever the last module passed to `Sequential` isn't a loss (namely of type `MSELoss`), as `forward` returns both an output and a loss, and there would otherwise be no way to compute a loss.

Moreover, `SGD`'s constructor takes as argument the parameters to optimize and the learning rate with which to perform the updates. An example run with a basic architecture is shown in V in listing 2.

III. Results

In `test.py` we build two models with two input units, two output units and three hidden layers of 25 units. Model 1 uses TanH as activation function, while Model 2 uses ReLU as activation function. The results are computed on 1000 epochs and shown on Table I. When looking at the accuracy, we can observe that Model 1 performs slightly better than Model 2. The variance of the accuracy of Model 2 is also slightly greater than that of Model 1. The variance is computed over 10 runs. The main purpose of this project is to develop a working framework and we see that our framework performs well with the instructed architecture.

Model	Accuracy	σ^2_{Test}
1	89.9%	0.03
2	90.0%	0.03

TABLE I: Results Table.

IV. Conclusion

Other losses, such as the *Mean-Absolute Erorr* (MAE) or *Cross-Entropy Loss*, could also be implemented. In this case, a generic base class of a loss would have to be defined from which all losses, including the `MSELoss`, would inherit. `Sequential` would also be adapted to accept a generic loss as last module. Similarly, there exist other optimization techniques that could also be implemented. A generic base class `Optimizer` would be defined and classes that implement different optimization techniques would inherit from it. All in all, our framework performs well, fully implements the requirements of the project, could easily be extended to implement additional machine learning methods and provides a clear and easy way to use it.

V. Appendix

Listing 2: Example run of our framework

```
# Assuming train_input, train_target,
    test_input and test_target given

# Imports
import modules
from optimizer import SGD

# Define architecture
model = modules.Sequential(
   modules.Linear(2, 25),
   modules.TanH(),
   modules.Linear(25, 2),
   modules.MSELoss())

# Define parameters
nb_epochs = 50
lr = 1e-3
optimizer = SGD(model.param(), lr)

# Train model
for e in range(nb_epochs):
   output, loss = model.forward(train_input,
       train_target)
   optimizer.zero_grad()
   grad = model.backward()
   optimizer.step()

# Test model
output, loss_test = model.forward(test_input,
    test_loss)
```

References

[1] S. Yegulalp, S. Yegulalp, and InfoWorld, "Facebook brings gpu-powered machine learning to python," Jan 2017. [Online]. Available: https://www.infoworld.com/article/3159120/facebook-brings-gpu-powered-machine-learning-to-python.html

[2] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NIPS-W*, 2017.

[3] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," *Journal of Machine Learning Research - Proceedings Track*, vol. 9, pp. 249–256, 01 2010.