Deep Learning Project 2
Mini deep-learning framework

Célia Benquet, Sven Borden & Artur Jesslen

***Abstract*—The first implementation of a deep learning framework from scratch can be seen as a rite of passage for deep learning engineers and researchers. The implemented framework wouldn't do as much as all the existing ones, or be as fast and as well-written. But it helps to have a better understanding of how a deep neural network and more specifically the backpropagation algorithm is working. Using only the Pytorch's tensor operations and the math library, we implemented a basic neural network framework. It provides some tools to build linear and 2D-convolutional networks, to run the forward and backward passes, supported by optimization tools.**

I. Introduction

Automatic differentiation (AD) consists in a set of techniques to numerically evaluate the derivative of a function specified by a computer program (Baydin et al., 2017 [1]). By applying the chain rule to sequences of elementary arithmetic operations and functions, it can automatically compute the derivatives by keeping track of the numerical values of the derivatives of the parameters along the propagation. A specialized counterpart of AD, the backpropagation algorithm, trains neural networks and compute a gradient descent to optimize an objective function. The gradients are obtained by the backward propagation of the error from a given output obtained through the forward pass. Pytorch, an open source machine learning library based on the Torch library (Paszke et al., 2017 [2]), provides its own version of AD of arbitrary scalar valued functions [3] with the `torch.autograd` module as well as a `torch.nn` module which provide us with a higher level API to build and train deep neural network. This project aims at designing a deep learning framework using only PyTorch's tensor operations and the math library and deactivating autograd to implement the forward and back passes ourselves as well as the neural network framework.

We implement the framework to build networks combining fully connected layers, convolutional layers, and either the Tanh or ReLU activation function. The parameters can be optimized using SGD for either Mean Square Error (MSE) or Cross Entropy losses. We also implemented the dropout and batch normalization regularization techniques and the max pooling feature extraction process.

II. Implementation

For our framework implementation, we aim to be the closest possible to the actual Pytorch implementation. Therefore, most functions that we implemented in our framework are interchangeable with similarly named functions in the `torch.nn` module. The general choices of implementation for our framework are presented, followed by a detailed explanation of the different modules. The heritage links between the different classes are presented as a diagram (Fig. 1)

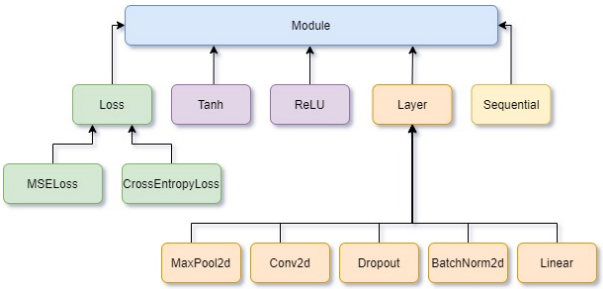

Figure 1: Heritage links between the implemented classes.

Parameter: The class *Parameters* is a very simple class that contains as attribute, the value of the parameter and its gradient that can be set to zero. This last one will be used for the weight optimization that we will see later.

Module: The superclass *Module* establishes the basis for all following sub-modules. *Module* contains the class attribute `training` that defines if the actual model is either in training or testing mode. It contains three dictionary attributes that store some useful variables. `_store` caches variables calculated during the forward path that can be reused to calculate the gradient. `_grad` caches the gradient with respect to each variable. That is why we use a dictionary. The key corresponds to the variable according to which the loss is derived. Finally, `_parameters` contains all the framework's sub-modules of a model and is used to set all the sub-modules' gradient to zero and to get the parameters of the model. Some abstract methods are also defined to enforce their definition in sub-modules. Those functions are `forward(*input)`, `backward(*gradwrtoutput)`, `zero_grad()`, and `local_grad()`. This last method is called during the forward path (by the `__call__` built-in method). It calculates the local gradient (i.e. the gradient with respect to the input) for a module and its result is

later used for the backward path. The two methods `save(path, filename)` and `load(path)` can also save and load pre-trained parameters in a model to/from a file (with extension .pt).

Neural Network Model: Similarly to the Pytorch implementation, for the user to create his own NN model, a class that inherits from the Module class can be created. To be functional, this class must implement the following method:

- `__init__()` that declares the neural network layers.
- `forward()` that implements the forward pass.
- `backward()` that implements the backward pass.
- Every other methods that one finds necessary (i.e training method).

An example of code to implement a neural network model can be found in the Appendix.

ReLU & Tanh: *ReLU* and *Tanh* implement the Rectified Linear Unit and the Hyperbolic Tangent. They inherit from the *Module* class. Therefore they redefine the methods forward(∗input) , `local_grad()` and `backward(*gradwrtoutput)` .

Loss: The class *Loss* establishes the basis for all type of losses and inherits from *Module*. It redefines the method `backward(*gradwrtoutput)` that is similar for all losses (i.e return the local gradient).

Mean Square Error and Cross Entropy Losses: *MSELoss* and *CrossEntropyLoss* implements the Mean Squared Error and the Cross Entropy losses. They inherit from the *Loss* class. Therefore they redefine the methods `forward(*input)` and `local_grad()` .

Layer: The class *Layer* establishes the basis for all type of layers and inherits from *Module*. It contains a new dictionary attribute `_params` that will store parameters of the layer (weights and bias). It implements an abstract method `_init_params(*args)` which initializes those parameters. It also redefines the `reset_grad()` which sets the gradient to zero for each parameter.

Linear: The class *Linear* inherites from *Layer* and implements the linear layer. The weights and bias initialization (in `_init_params()` is done using random uniform values between $\pm\frac{1}{\sqrt{dim\_in}}$, with *dim_in* being the dimension of the feature vector before the linear layer. Naturally, the class also redefines `forward(*input)` , `local_grad()` and `backward(*gradwrtoutput)` .

2D Convolution: The class *Conv2d* also inherits from *Layer* and implements the 2D convolutional layer. The weights initialization (`_init_params()`) is done using the method described in Glorot & Bengio [4] also known as the *Xavier initialization*. The bias are set to zero. Convolutional layers also support non-squared kernel, stride and zero-padding. It redefines `forward(*input)` and `backward(*gradwrtoutput)` but not `local_grad()` as precomputing a local gradient was less convenient. The convolutions has not been optimized using Fast Fourier Transform (FFT) and are therefore very slow to compute.

Batch Normalization: The class *BatchNorm2d* inherits from *Layer* and implements the batch normalization layer by following the model described in Ioffe et al. (2015 [5]). The parameter γ is initialized to $\mathbb{1}$ while the second parameter β is set to zero. The class redefines the same functions as for the Conv2d layer. The batch normalization back-propagation formula has been largely inspired by the formula described by Clément Thorey on his GitHub page [6]. During training, it learns the empirical moments of the input, which are re-used during test (single test sample statistics are not available).

Max Pooling: The class *MaxPool2d* inherits from *Layer* and implements the 2D max pooling layer. It can take as parameter a (non-)squared kernel. As for the other sub-classes of *Layer*, it redefines `forward(*input)` , `local_grad()` and `backward(*gradwrtoutput)` .

Dropout: The class *Dropout* inherits from *Layer* and implements the dropout layer as described in Srivastava et al. (2014 [7]). We define the two parameters `p` (probability of dropping a unit) and `inplace` (modify the input if True). Naturally, it also redefines `forward(*input)` , `local_grad()` and `backward(*gradwrtoutput)` . As for BatchNorm2d, it acts differently during train and test.

Sequential: The class *Sequential* inherits directly form *Module* and implements sequential layers. It contains an attribute `layers` which is a List of all layers contained in the sequential module. For each function (`reset_grad()` , `forward(*input)` , `local_grad()` , `backward(*gradwrtoutput)`), it makes sure that it is called for all layers by calling them in the reversed order.

III. Results

Our framework is able to train a model, as one can see a decreasing error rate through the epochs on Figure 2. The dataset is composed of 2 classes. The separation curve corresponds to a circle of radius 0.5, with class 0 being outside and class 1 inside the circle. From Figure 3, its predictions seem accurate and all misclassified points are located on the border of the class regions.

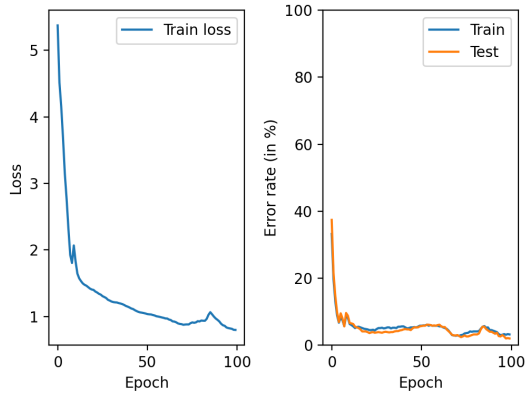

Figure 2: Train loss and train and test error rates during the training of a linear model. *Model:* Linear(2, 25), ReLU(), Linear(25, 25), ReLU(), Linear(25,25), ReLU(), Linear(25,2).

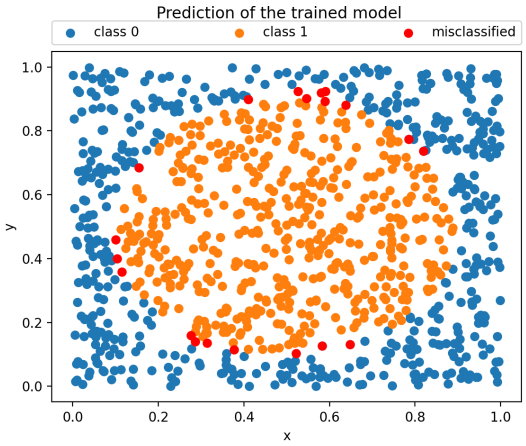

Figure 3: Prediction of the linear model, misclassified points are represented in red.

IV. Comparison with torch.nn & torch.autograd

Our framework aims to be the closest possible to the Pytorch implementation of the neural network module. The main divergence between Pytorch and our framework consists in the usage of graphs to compute the backward pass in the Pytorch implementation. Consequently, our current implementation suggests that the corresponding backward method must be explicitly written by the user when creating a net as illustrated in the Appendix.

Furthermore, another limitation is that the convolutions are not optimized. The use of for loops are dramatically slowing down the convolutions calculations. FFT would have been a lot more adapted to do that kind of calculations and could be a further implementation in our framework.

Appendix

```
import torch
import framework as ff

class Net(ff.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.layer = ff.Sequential(
            ff.Linear(10, 200),
            ff.ReLU(),
            ff.Linear(200, 10))

    def forward(self, x):
        x = self.layer(x)
        return x

    def backward(self, criterion):
        d = criterion.backward()
        d = self.layer.backward(d)
        return d

    def train_(self, data, target, \
            criterion=ff.MSELoss(), \
            eta = 1e-5):
        pred = self(data)
        loss = criterion(pred, target)
        self.zero_grad()
        self.backward(criterion)
        for p in self.parameters():
            p.p -= eta * p.grad
        # Do your training stuff here
```

References

[1] Atılım Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *J. Mach. Learn. Res.*, 18(1):5595–5637, Jan 2017.

[2] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. 2017.

[3] Automatic differentiation package - torch.autograd — PyTorch master documentation, May 2020. [Online; accessed 16. May 2020].

[4] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. 2010.

[5] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *Cornell University*, 2015.

[6] What does the gradient flowing through batch normalization looks like ?, Jan 2016. [Online; accessed 22. May 2020].

[7] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.