Mini Deep Learning Framework
Deep Learning Mini Project 2 Report

Ravinithesh Annapureddy, Anmol Prasad, Kushagra Shah
École Polytechnique Fédérale de Lausanne, Switzerland

I. Introduction

The aim of the project is to create a neural network framework (DL framework) using only PyTorch's tensor operations and the standard math library, specifically without using autograd or the neural-network modules provided by any other library. Our framework provides the tools to:

- build networks combining fully connected layers, Tanh, ReLU and Sigmoid activation's.
- run the forward and backward passes to perform prediction and train the weights respectively.
- optimize parameters with Stochastic Gradient Descent (SGD) using Mean Squared Error (MSE) between the predicted value and the actual value.

The rest of the report is organized as follows: Section 2 lays out the structure implemented for the framework, section 3 describes the functionalities of the framework, section 4 describes the experiments performed using the developed framework which is followed by presenting the results of the previous experiments. Summary and future directions are given in the last section.

II. Framework

In this section, we sketch the design of the framework and elaborate the implementation of each component. The framework is mainly composed of 6 components. While one component describes the structure of the base modules used to build the other modules, the rest of the components contain the fully connected layer, the optimizer, functions to apply activation and loss, and a module to encompass the model created using the various layers sequentially.

A. Base Modules

This subsection describes the two base classes that are created for the project which act as superclasses and are inherited by the subclasses that implement the functionalities mentioned in Section I.

1) Module: Module base class is inherited by all the subclasses except for the optimizer. Its implementation is shown below:

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

Each method performs the following tasks

- The *forward* method accepts a tensor and returns a tensor after applying operations in the forward pass of that module.
- The *backward* method gets as input a tensor containing the gradient of the loss w.r.t the module's output, accumulates the gradient w.r.t. the parameters, and returns a tensor containing the gradient of the loss w.r.t. the module's input.
- The *param* method returns a list of tuples of the parameters of the module i.e. the tuple of parameter and gradient tensors for weight and bias of the module. For the module without parameters like the activation functions or loss, an empty list is returned.
- The *zero_grad* method sets the gradient tensors of the module to zero in those modules that have a gradient tensor.

2) Optimizer: Optimizer base class is inherited by the subclass that implements a Stochastic Gradient Optimizer in the project. Its implementation is shown below:

```
class Optimizer(object):
    def step(self):
        raise NotImplementedError
```

The class contains only one method, *step*, that updates the weight parameter using the gradient of the w.r.t its output and the learning parameter (see Section III-E).

III. Methods

In this section, we describe each of the designed modules to perform a specific function.

A. Layers: Linear

One of the crucial components of a Deep Neural Network is the inclusion of hidden layers. For this project, we have implemented a fully connected linear layer in which each unit of the previous layer is connected to all the units of the next layer. The methods of the linear module are the following:

- *__init__(self, in_features, out_features, bias, weightsinit)* method initializes the layer by creating a weight tensor of size in_features $\times$ out_features using the weight distribution described using the *weightsinit* parameter. *in_features* and *out_features* are integers that indicate the

number of input and output units of the layer respectively. *bias* is a boolean argument that is *True* by default indicating whether the layer should train an additive bias or not and they are initialized to zero of size equal to the out_features, if *True*. *weightsinit* can accept three possible strings

  - *uniform*: This is the default initialization similar to PyTorch, where the weights are initialized from a uniform distribution in the range $(-k, k)$ where $k = \sqrt{1/in\_features}$.
  - *xavier*: The weights are initialized using Xavier-normal initialization, which is from a normal distribution in with mean 0 and standard deviation of $\sqrt{(2/(in\_features + out\_features))}$. This type of initialization is preferred when using *TanH* activation for the layers.
  - *kaiming*: The weights are initialized using Kaiming-normal initialization, which is from a normal distribution in with mean 0 and standard deviation of $\sqrt{2/in\_features}$. This type of initialization is preferred when using *ReLU* activation for the layers.

  Additionally, the module keeps track of the gradient of weights and biases by storing those tensors.
- *forward* method computes the output of layer l as following:

$$z^{[l]} = w^{[l]}x^{[l-1]} + b^{[l]} \tag{1}$$

  where $w^{[l]}$ and $b^{[l]}$ are the weights and biases of the layer and $x^{[l-1]}$ is the output of the previous layer's activation function (or the input if l = 1).
- *backward* method computes gradient with respect to the output as following:

$$\begin{aligned} dw^{[l]} &= dz^{[l]}x^{[l-1]}, \\ db^{[l]} &= dz^{[l]}, \\ dx^{[l-1]} &= {w^{[l]}}^T dz^{[l]} \end{aligned}$$

  where $w^{[l]}$ are the weights of the layer, $dz^{[l]}$ is the gradient backpropagated from layer $l+1$, $dw^{[l]}, db^{[l]}$ are the gradients of the weight and bias tensors and $dx^{[l-1]}$ is the derivative of loss w.r.t the output.

B. Activation Functions

In this framework, we have implemented three activation functions that are applied to the output of the linear layer. Each of them inherits the module class, define the forward and backward methods as described below and store the output of the forward pass to be used in the backward pass.

1) TanH:

- *forward* method applies inbuilt tensor operation of the Hyperbolic Tangent function.
- *backward* method returns the derivative of TanH as following: $gradwrtoutput \times (1 - out^2)$, where out is the output of the forward method and gradwrtoutput is the gradient w.r.t the output.

2) ReLU:

- *forward* method sets those entries of the input tensor that are less than zero to zero.
- *backward* method returns the derivative of ReLU as following: $gradwrtoutput \times (out)$, where out is transformed tensor of the input in which the entries less than zero are set to zero and those that are greater than zero are set to one. *gradwrtoutput* is the gradient w.r.t the output.

3) Sigmoid:

- *forward* method applies inbuilt tensor operation of the Logistic function.
- *backward* method returns the derivative of Sigmoid as following: $gradwrtoutput \times (out - out^2)$, where out is the output of the forward method i.e. the sigmoid of the input tensor to the module and gradwrtoutput is the gradient w.r.t the output.

C. Sequential

To combine the forward and backward computations and parameter updates of the multiple linear layers and the activation layers, we created a sequential module. The methods of the module are the following:

- The module is initialized by passing a list of layers and activations in the same order to be considered to build the model.
- *forward* method executes the forward method for each module and passes the outputs to the next module.
- *backward* method similarly executes the backward for each module in reverse order and propagates the gradients backwards.

An example to define a model is shown below. It includes an input layer and output layer with two units each and 3 hidden layers with 25 units each. It uses TanH activation for all the layers except the last layer which uses Sigmoid activation.

```
Model = Sequential(Linear(2, 25), TanH(),
                   Linear(25, 25), TanH(),
                   Linear(25, 25), TanH(),
                   Linear(25, 25), TanH(),
                   Linear(25, 2,),
                   Sigmoid())
```

D. Loss: Mean Squared Error

To obtain the error between the predicted value of the network and the target value, we implemented the MSE loss which helps in updating the parameters of the layers. The methods of the module are the following:

- *forward* method accepts the *outputs* i.e. the prediction tensor of the network and the *targets* i.e. the desired prediction from the training data and returns the loss that is defined as $\frac{1}{N} * \sum (error)^2$, where N is the number of predicted values and $error = output - target$ for each sample. The error value is stored for the backward pass calculation.
- *backward* method returns the derivative of Mean Squared Error as following $\frac{2}{N} * \sum error$.

E. Optimizer: Stochastic Gradient Descent

We have implemented the SGD algorithm to update the weights of the layers during training. The module inherits the *Optimizer* class and the implemented methods are the following:

- The module is initialized by passing a list of parameters tuples i.e. the parameter and the gradient tensor tuple and learning rate.
- *step* method updates the weights and bias parameters using the respective gradients as following:

$$W - (\eta * \delta W) \tag{2}$$

where W is weight/ bias parameter, η is learning rate and δW is the gradient w.r.t the output of the layer.

IV. Experiments

To test our implementation of the framework, we define four models to experiment with various parameters. All models have an input layer, an output layer and 3 hidden layers with 25 neurons each. The factors that can be used for experiments are the type of activation and weight initialization, thus creating four possibilities (see Table I).

A. Data

A custom function is written to generate a toy dataset to train and test the developed framework. The custom function generates a dataset of user-specified size, sampled uniformly in $[0, 1]^2$, each with a label 0 if the data-point is outside the disk centred at $(0.5, 0.5)$ of radius $1/\sqrt{2\pi}$, and 1 if it is inside. We generate a set of 1000 points each, for training and testing the networks.

B. Experiment Parameters

Each model is trained for 1000 epochs with a learning rate of 0.01 and a mini-batch size of 100. We repeat the training for 10 iterations by re-initializing the models with random weights to obtain error estimates.

V. Results

A. Loss

The loss after each epoch for each of the 10 iterations, along with the average loss across all the iterations, is shown in Figure 1 for the four possible models. As expected, we observe that the loss decreases with the epochs while training the neural network. The loss curve drops sharply when using specific weight initialization for the activation function used in the model in comparison with uniform weight sharing. Using Xavier/Kaiming weight initialization, helps sustaining the variance of the activations and back propagated gradients in the deep layers when using the TanH/ReLU activations. This prevents the exploding/vanishing gradient problem in the backpropagation phase.

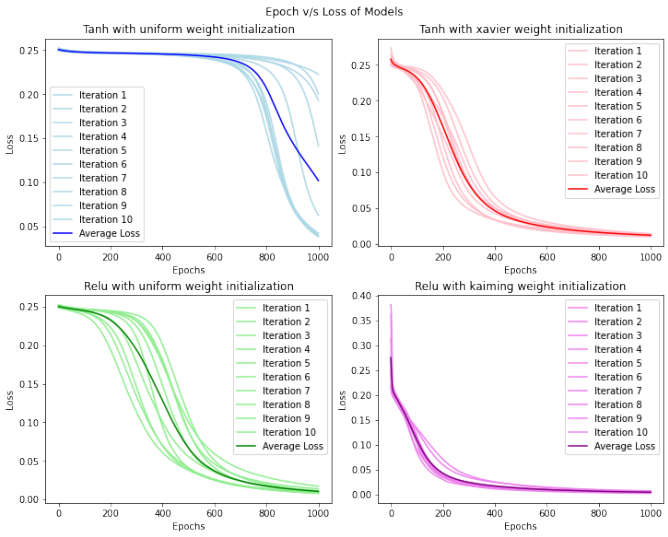

Fig. 1. Loss across epochs over 10 iterations

B. Errors

The errors produced on testing the trained models on the test set are presented in Table I. Drawing on similar lines from the loss, the test errors rates when using TanH and ReLU activation are lowest when using Xavier and Kaiming weight initialization respectively.

Model	Activation	Weight Init.	Train Error	Test Error
1	TanH	Uniform	1.90%	1.90%
2	TanH	Xavier	0.80%	1.20%
3	ReLU	Uniform	0.60%	1.60%
4	ReLU	kaiming	0.40%	1.40%

TABLE I
TEST ERRORS OF THE FOUR MODELS

VI. Conclusion

With the availability of popular Deep Learning Frameworks like PyTorch, Tensorflow and Keras, and especially the automatic differentiation functions, many users can now construct and work on Deep Learning models more easily. However, not many understand the basic mechanisms such as the backpropagation algorithm. Through this project [1], we were able to implement backpropagation in a sequential structure with a linear layer using multiple activation functions, and optimize using SGD employing the MSE loss. Additionally, the generic structure of the modules allows easy expansion of the framework to include new features such as Batch Normalization, Dropout layers, Convolutional layers, Cross-Entropy Loss etc.

References

[1] "UNIGE 14x050 – EPFL EE-559 – Deep Learning." [Online]. Available: https://fleuret.org/dlc/