Deep Learning Project 1
- Mini deep learning framework -

Deep Learning (EE-559)
École Polytechnique Fédérale de Lausanne

Salim Ben Ghorbel
salim.benghorbel@epfl.ch

Ahmed Ben Haj Yahia
ahmed.benhajyahia@epfl.ch

Nour Ghribi
nour.ghribi@epfl.ch

***Abstract*—Implementing modern deep learning models relies heavily on proper frameworks and API's as they help minimizing code duplication and maximising ease of use as well as good structure. TensorFlow and PyTorch are ones of the well known platforms for deep learning. To appreciate this aspect, in this project, we implemented our own mini-framework and compared our performance with PyTorch's Neural Network (NN) library, from which it was inspired.**

I. Introduction

We will start by explaining the high level aspects of deep learning and the intuition behind neural networks, to understand better the use of our framework. In a second time, we will give some details about the implemented modules, loss function and optimizer used in this framework. Finally, we will build a model using our implementation, train it and evaluate its performance.

II. Overview

Deep Learning is mainly the training of big Artificial Neural Networks of "deep" stacks of parameters. The idea behind a neural network (NN) is to simulate lots of densely interconnected "cells" called units so you can get it to learn things, recognize patterns, and make decisions in a "human-like" way.

When learning or operating normally, patterns of information are fed into the network via the input units, which trigger the layers of hidden units, and these in turn arrive at the output units. This design is called a feed forward. Each unit receives inputs from the units to its left, and the inputs are multiplied by the weights of the connections they travel along (Linear modules). Every unit adds up all the inputs it receives in this way and if the sum is more than a certain threshold value, the unit "fires" and triggers the units to its right it is connected to (Activation functions). [1]

The ultimate goal being to find the best possible approximation function that relates each input to its correct output, determining this function is an optimization problem that is treated using a feedback process called back propagation.

The idea is to compute the output produced by the NN for an input (forward pass), use a loss function to evaluate how far is the predicted output from the target, and update the parameters of the NN in order to decrease this loss.

Following this intuition, our framework builds neural networks using class Sequential, representing an MLP (Multi Layer Perceptron) that combines multiple linear and non linear modules. These modules have at least a forward and a backward method each to compute outputs and learn through back propagation respectively. Learning is done by calculating the loss and its gradient, accumulating the gradients by a backward pass, then optimizing using standard gradient techniques - stochastic Gradient Descent (SGD) in our case. Further details are given in the foloowing sections.

III. Framework

The main base class around of which revolves the framework is the Module class. One whole neural network can be seen of as an ordered collection of such modules. The module base class forces the classes that inherits from it to redefine three main methods (forward, backward, param). The forward method deals with activations and feeds forward, whereas the backward method deals with gradients and back propagation. We defined one other base class, Initializer to handle parameter initialization, and implemented a Xavier initializer. [2]

We can separate the implemented modules into 3 main categories:

- **Linear Module**: Describes a dense layer with input dimension and output dimension. The weights initialization is done using Xavier initialization.
- **Sequential**: This is the building module for our MLP, it takes multiple modules as parameters and creates a neural network in the order of the parameters given.
- **Non-Linear Modules (activation functions)** Non-Linear modules are the activation functions to use after each dense layer.

A. Linear Module

Each layer in a neural network has learnable parameters. The forward method depends on these parameters that we initialize using Xavier initialization in order to avoid the layer activation outputs from exploding or vanishing. In the backward method we calculate the derivatives and store the gradient for these parameters.

- **Instantiation Parameters:** dim_{in}, dim_{out}
- **Learnable Parameters:**
  $W \in \mathbb{R}^{dim_{out}} \times \mathbb{R}^{dim_{in}}, b \in \mathbb{R}^{dim_{out}}$

- **Forward**($X \in \mathbb{R}^{dim_{in}}$): output: $\in \mathbb{R}^{dim_{out}}$
- **Backward**($Z \in \mathbb{R}^{dim_{out}}$): output: $\in \mathbb{R}^{dim_{in}}$

B. Non-Linear Modules (activation functions)

Given a loss function $\mathcal{L}$, input x, output y and activation function f, we are interested in $\frac{\partial \mathcal{L}}{\partial x}$ given that $y = f(x)$:

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial x}$$

1) **ReLU**: $y = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$

   the gradient is: $\frac{\partial \mathcal{L}}{\partial x} = \begin{cases} \frac{\partial \mathcal{L}}{\partial y} & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$

2) **LeakyReLU**: $y = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases}$

   With the gradient: $\frac{\partial \mathcal{L}}{\partial x} = \begin{cases} \frac{\partial \mathcal{L}}{\partial y} & \text{if } x > 0 \\ \alpha \frac{\partial \mathcal{L}}{\partial y} & \text{otherwise} \end{cases}$

   α is the negative slope that we define by default $\alpha = 0.01$.

3) **Tanh**: $y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

   With gradient: $\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y}(1 - tanh(x)^2)$

4) **Sigmoid**: $y = \sigma(x) = \frac{1}{1+e^{-x}}$

   Wich will give gradient: $\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y}\sigma(x)(1 - \sigma(x))$

All the loss functions defined inherit from module but without defining the param method as they have no parameters, thus param will return an empty list on them.

C. Sequential

This module, is the building block for all neural networks using our framework.

- **Instantiation Parameters:** *modules*
  The Sequential organizes the modules - the l (layers) and activation functions - passed as arguments in a sequential (ordered) manner.
- **Forward**($X \in \mathbb{R}^{dim_{in}(1)}$): $\in \mathbb{R}^{dim_{out}(l)}$
  Iterates over all the layers, in an ascending order, applying the forward function of the current module on the output from the previous module. (1 to l)
- **Backward**($X \in \mathbb{R}^{dim_{out}(l)}$): $\in \mathbb{R}^{dim_{in}(1)}$
  Iterates over all the layers, in a reversed order, applying the backward function of the current module on the output from the previous module. It yields the gradient of the loss with respect to the paramaters.

D. Loss

The losses inherit from module and define how to compute their respective loss in forward and their derivatives in backward and feed it to the backward method of the model used in the instantiation.

- **Instantiation Parameters:** The model
- **Forward**(Prediction, target $\in \mathbb{R}^{dim_{out}(l)}$)
- **Backward** Compute the derivative and initiate back propagation on the model.

E. Optimizer

Gradient descent is used for optimizing (minimizing) an objective function $\mathcal{L}(\theta)$ with $\theta \in \mathbb{R}^D$ being the model parameters vector. The basic principle is to update the weights (parameters) of a given model by taking a step in the opposite direction of the gradient of the objective function with respect to θ. Various algorithms and variations of gradient descent exist to to optimize this approach and avoid getting trapped in sub optimal local minimum. We chose to implement ***Stochastic Gradient Descent (SGD)*** which is a variation of gradient descent that uses only a certain training input (x_i, y_i) or a batch of inputs $(x_i, y_i)_{i \in \{1,n\}}$ to update.

Stochastic Gradient Descent rule, given μ the learning rate:

$$\theta^t = \theta^{t-1} - \mu \nabla_\theta \mathcal{L}(\theta; x_i, y_i)$$

Batched SGD version:

$$\theta^t = \theta^{t-1} - \mu \nabla_\theta \mathcal{L}(\theta; x_{i,i+n}, y_{i,i+n})$$

In our implementation, optimizers inherit from module and define a step function that updates the gradient for all the weights of the model using the derivatives accumulated in the backward pass. We defined *Stochastic Gradient Descent* using this scheme, defined a method zero_grad() that resets the gradient to 0.

- **Instantiation Parameters:** The model

IV. Data Generation

For evaluating our framework, we set up the task of training a simple neural network - Multi Layer Perceptron - to classify two-dimensional objects into one of two classes (1,0).

For our training dataset, we generated a set of 1000 points sampled uniformly in $[0, 1]^2$, each with label 0 if outside the disk of radius $\frac{1}{\sqrt{2\pi}}$ and 1 inside.

For our test dataset, we generated a set of 1000 points with the same scheme. (Figure 1)

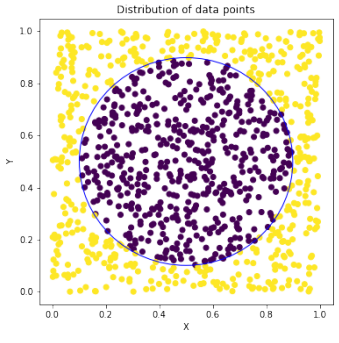

Fig. 1. Data used for training/testing

V. Neural Network using our framework:

To test our framework we need to build a model using the defined modules and evaluate its performance. We will build a neural network using our implementation and an identical model using PyTorch's framework to compare the two.

A. *Architecture Description*

We built a network with two input units, three hidden layers of 25 units and two output units to test our framework. (Figure 2)

It's defined as a sequential model with:

1) Input Layer of 2 units followed by LeakyReLU activations.
2) Fully connected Layer of 25 input units, 25 output units followed by LeakyReLU activations.
3) Fully connected Layer of 25 input units, 25 output units followed by LeakyReLU activations.
4) Output of 2 units followed by Sigmoid activations.

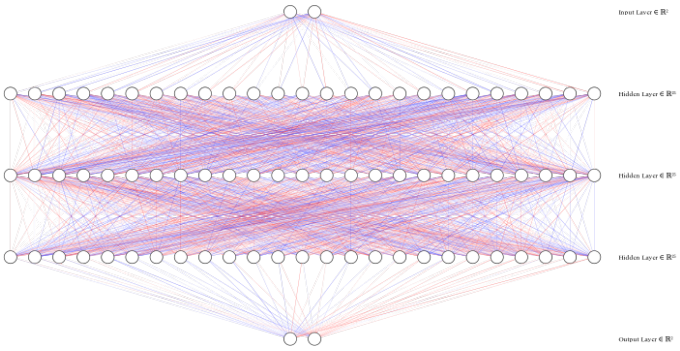

Fig. 2. The implemented model

B. *Loss calculation and Optimizer:*

For the loss calculation, we used our implemented module LossMSE which is used as Criterion in the forward and backward pass as described in the table below.

Loss	Forward pass	Backward pass
MSE	$\frac{1}{n}\sum_{n=1}^{n}(y_i - \hat{y_i})^2$	$2(Y - \hat{Y})$

We used our implemented SGD optimizer module in each training iteration i.e first we set the gradients of all our model's parameters to zero (using zero_grad()), then after doing the backward pass (loss.backward()) to accumulate the derivatives, we perform a stochastic gradient descent step.

C. *Hyperparameter tuning*

Using a grid search algorithm, we tuned the learning rate by running our model with different values of learning rates $\mu \in [0.01, 0.015, 0.04]$

The resulting mean squared errors are shown in figure 3 taking both our framework model and using PyTorch.

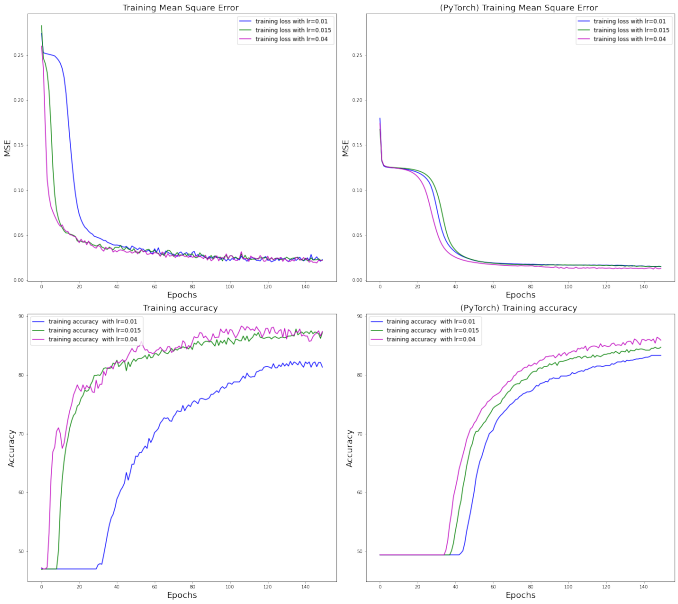

Fig. 3. Hyperparameter tuning

D. *Model Performance:*

To better evaluate our model's performance using the created framework, we created and trained the same described model using the PyTorch framework. This will enable us to compare these two, and asses our work.

In figure 3, we can see that the progress of the accuracy using PyTorch started later than in our model (roughly 40 epochs against 20 epochs approximately). We can also see that, using our framework, the model with learning rates 0.01 and 0.15 surpassed the one with PyTorch but converge almost to the same values. On the other hand, there is a significant difference between the two frameworks when using 0.04 as learning rate.

Framework	Loss	Accuracy
Ours	0.0227	81.5%
PyTorch	0.0149	83.8%

VI. Summary:

All in all, this project was instructive to discover how the internal modules and functions of deep learning frameworks such as PyTorch work and made us appreciate their availability for modern machine learning problems.

References

[1] https://www.explainthatstuff.com/introduction-to-neural-networks.html
[2] Weight Initialization in Neural Networks: A Journey From the Basics to Kaiming