Deep Learning Project 2
- Deep Learning Framework -

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

Amir Ghali
amir.ghali@epfl.ch

Mahmoud Sellami
mahmoud.sellami@epfl.ch

Ahmed Ben Haj Yahia
ahmed.benhajyahia@epfl.ch

***Abstract*—This report details our implementation of a "Deep Learning framework", using Pytorch as an inspiration and as a model. This framework allows to perform a simple classification task, by implementing linear and non-linear modules, a loss function and an optimizer to allow build, train and test the accuracy of a multi layer perceptron from scratch.**

I. Introduction

We will start by giving a general understanding of deep learning and neural networks to get some intuition and help understand the aim of this framework and how it actually works. In a second time, we will give some details about the implemented modules, loss function and optimizer used in this framework, designed specifically for binary classification. Finally we will talk about how we used this implementation to build a model, train it and evaluate its performance.

II. Overview

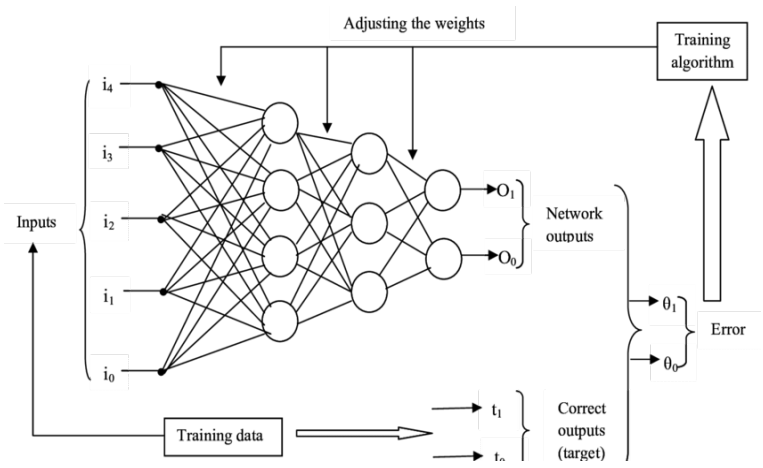
The basic idea behind a neural network (NN) is to simulate lots of densely interconnected "brain cells" called units so you can get it to learn things, recognize patterns, and make decisions in a human-like way.

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 feedforward network. 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 backpropagation.

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. A convenient update is generally computed using variants of the Gradient Descent algorithm. The computation of gradients is called the backward pass.

Following this intuition, we designed in our framework a class Sequential, consisting of an MLP (Multi Layer Perceptron) that combines multiple linear and non linear modules (Class Module). 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 (Class MSELoss) and its gradients, then optimizing using standard gradient techniques (Class SGD).[2]Further details are given below.



III. Modules

A. Linear Module

Considering an input variable x and an output y, a linear function can be expressed as y = W*x + b, where W(weights) and b(bias) are the parameters to learn. With $\mathcal{L}$ being our loss, the expressions of the gradients are as follows:

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

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial W} = \frac{\partial \mathcal{L}}{\partial y}X^T$$

$$\frac{\partial \mathcal{L}}{\partial b} = \sum_{i=1}^{N}(\frac{\partial \mathcal{L}}{\partial y_i})$$

B. Non-Linear Modules

1) *ReLU:*

$$y = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Activation function (No parameters to be optimized over). It's gradient over the loss is defined as:

$$\frac{\partial \mathcal{L}}{\partial x} = \begin{cases} \frac{\partial \mathcal{L}}{\partial y} & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

2) *Tanh:*

$$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Also an activation function, with gradient:

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

IV. Loss Function

A loss function is a measure of how good the prediction model does in terms of being able to predict the expected outcome. We convert the learning problem into an optimization problem, define a loss function and then optimize the algorithm (By updating the parameters) to minimize it (Using the gradients of the loss as seen above).

In our framework we implemented the Mean Square Error (MSE) loss function: MSE is sensitive towards outliers, and given several examples with the same input feature values, the optimal prediction will be their mean target value. MSE is thus good to use when the target data, conditioned on the input, is normally distributed around a mean value.
We have :

$$\mathcal{L}(\hat{y}, y) = \frac{1}{N}\sum_{i=1}^{N}(\hat{y_i} - y_i)^2$$

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{2}{N}\sum_{i=1}^{N}(\hat{y_i} - y_i)$$

Where $\hat{y_i}$ is the predicted output value and y_i is the target value.

V. Optimization Algorithm

Once we made the forward pass and evaluated the loss, we'll need this information to update the parameters and optimize our model. In our framework we implemented the Stochastic Gradient Descent (SGD) optimizer. While in standard Gradient Descent (GD), we have to run through all the samples in the training set to do a single update for a parameter in a particular iteration, with SGD on the other hand, we use only one, or a subset(Mini-batch SGD in this case), training sample to do the update.

Thus, if the number of training samples is very large, using standard gradient descent may take too long because in every iteration we are running through the complete training set. On the other hand, using SGD will be faster because we only use one training sample and it starts improving itself right away.

SGD often converges much faster compared to GD but the error function is not as well minimized as in the case of GD. Nevertheless, the parameters approximations we get from SGD are more than enough because they reach the optimal values.
In case of GD:
$w = w - \eta\nabla\mathcal{L}(w) = w - \eta\frac{1}{N}\sum_{i=1}^{N}\nabla\mathcal{L}_i(w)$

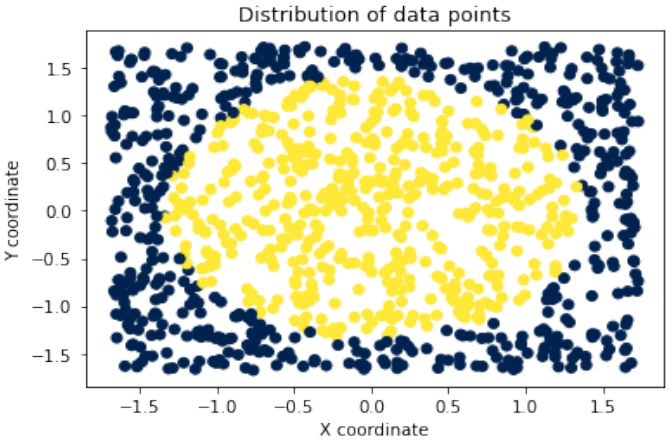
In case of SGD:
$w = w - \eta\nabla\mathcal{L}_i(w)$

Where η is the learning rate.

VI. All together

We create a neural network by calling our implemented class Sequential, we feed it with linear(Fully connected layer) and non linear(ReLU and Tanh) modules as arguments. These modules have each a forward method to perform the forward pass, a backward method that uses gradients to do the backward pass, and a param method that returns back the parameters(only for linear modules, None for activation functions). After creating the network and building the modules we train the model by doing the forward pass, computing the loss then performing the backward pass. We then update our weights with the optimizer's step function, and we keep doing this for all the data points for as many epochs as told. This is implemented in the train-model function, and after that we test our model's performance on new test samples to evaluate its accuracy on unseen data.

1) *Data:*
Our data consists of a 1,000 data points sampled uniformly in [0,1]², each with a label 0 if outside the disk centered at (0.5,0.5) of radius $\frac{1}{\sqrt{2\pi}}$, and 1 inside.



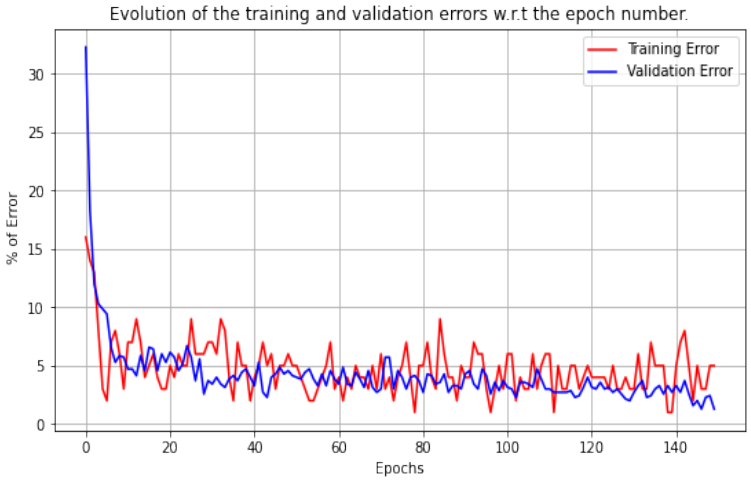
2) *Model:*
We used the suggested model, which consists of two

input units, two output units, three hidden layers of 25 unit. We chose the activations that gave us the better performance overall. Here is the code:

```
Input_Units = 2
Output_Units = 2
Hidden_Units = 25

model = m.Sequential(m.Linear(Input_Units,Hidden_Units),
                     m.ReLU(),
                     m.Linear(Hidden_Units,Hidden_Units),
                     m.ReLU(),
                     m.Linear(Hidden_Units,Hidden_Units),
                     m.Tanh(),
                     m.Linear(Hidden_Units,Output_Units),
                     m.Tanh()
                     )
```

3) *Results:*
These were our results for train and validation data over 150 epochs:



and the final error rate on test data was 3%

VII. Summary:

Inspired from the Pytorch module, this was our way to implement from scratch the modules required to build, train and test a neural network. For a better performance and future designs, one can start by thinking of new ways to re-implement the the optimization algorithms and backward methods, but also to try more activation functions(Signum, Sigmoid..), more loss functions(Cross-Entropy, MAE, Huber..) and other optimizers(NAG, Adam, Adagrad..)

References

[1]https://www.explainthatstuff.com/introduction-to-neural-networks.html
[2]https://www.researchgate.net/figure/How-a-neural-network-works_fig1_308094593