RocketDeepL

A mini deep learning framework

Francis **Damachi** - Costanza **Volpini**

***Abstract*—In the last few years, Neural Networks have proved to be one of the most effective solutions when approaching several different kinds of problems (e.g. image and speech recognition, language processing). The aim of this project is to design a multi-layer perceptron capable using the standard math library and the basic tensor operations of Pytorch.**

I. Introduction

A neural networks is composed by a series of hidden layers, an input layer with its input neurons (or nodes) and an output layers with its output neurons (see Fig. 1).

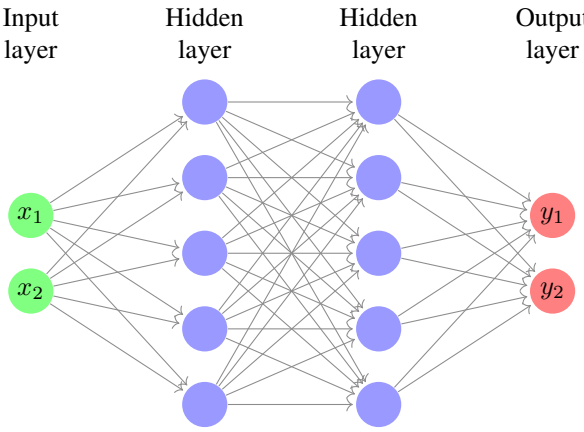

Fig. 1: General neural network with two hidden layers.

Perceptron is the key concept of a neural network, it takes some inputs and it produces a binary output. Some *weights* are then introduced, showing the importance that each input has in respect to the output. The output is then computed looking if the value obtained $\sum_i w_i x_i$ where w_i is a weight and x_i is an input neuron is less or greater than a threshold value. *Bias* could be seen as "how much it is probable that the perceptron will output a 1". Then the output will be calculated as followed:

$$\text{output} = \begin{cases} 1 \text{ if } \sum_i w_i x_i + b \geq 0 \\ 0 \text{ otherwise} \end{cases}$$

To make our algorithm learn we need to use *activation neuron* (see Sec. V), then the output would not be 0 or 1, but instead would be $\sigma(wx + b)$. It is important to notice that σ is a non-linear function. These non-linear function guarantee a better accuracy of the model.

In our implementation, weights and bias are initialized from a uniform random distributed and the values are defined by $[-std, +std]$ (where std is the standard deviation) as explained in Section II. In case of a binary classification our neural network will provide an accuracy of 50%. The main core of the project could be find in the pipeline where we let the neural network learn. First of all we apply the *forward pass*, then *backward pass* and for last we update all the *parameters*.

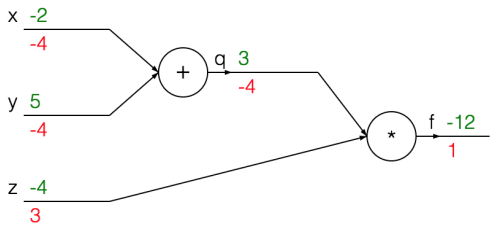

Fig. 2: Example of forward and backward passes. The first is represented in green and the second in red. [**??**]

1) *Forward pass*: computes values from inputs to output.
2) *Backward pass*: start from the output and performs backpropagation applying recursively chain rule in order to compute all the gradients of the loss until the input. This loss function is used in order to detect how much the output differs from the expected one.
3) *Update parameters*: to minimize the train loss we update all the parameters looking on the gradients found during the *backward pass*.

In the following sections we will analyze deeply all the layers and we will provide a description of the structure of the framework code. Let define $\left[\frac{\partial l}{\partial s^{(l)}}\right] = \left[\frac{\partial l}{\partial x^{(l)}}\right] \odot \sigma'(s^{(l)})$.

Name	**Formula**
Forward pass	$x^{(l)} = \sigma(w^{(l)} x^{(l-1)} + b^{(l)})$
Backward pass	$\left[\frac{\partial l}{\partial w^{(l)}}\right] = \left[\frac{\partial l}{\partial s^{(l)}}\right] \left(x^{(l-1)}\right)^T$, $\left[\frac{\partial l}{\partial b^{(l)}}\right] = \left[\frac{\partial l}{\partial s^{(l)}}\right]$

II. Data set

Data used for train and test are composed by 1000 points sampled distributed in range $[0, 1]^2$, points outside the radius $\frac{1}{\sqrt{2\pi}}$ are labelled with 0 target, otherwise 1 (See Fig. 3). Features can also be encoded using a one-hot encoding scheme. The code that provide the generation of the dataset can be found in the file `generator_training_set.py`.

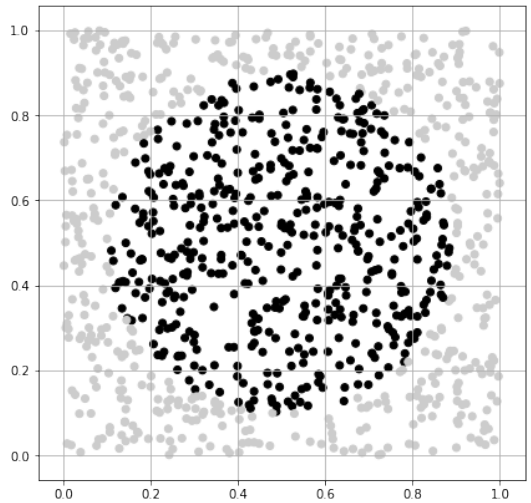

Fig. 3: Points are sampled distributed in range $[0, 1]^2$. Points labelled with 0 are shown in gray, else in black.

III. Implementation

We have decided to use a modular approach in order to make possible to modify or expand it. Backpropagation was implement storing all the information of the graph, in order to do that each node keeps tracks of its input other than calculating its result. Then to compute the derivative of its result, we can just multiply its value with the current gradient and pass it to its inputs. The forward pass just compute the output and return it, instead for the backward pass each node receive the gradient from its output (`gradientwrtoutput` parameter) and it multiplies this value with the gradient of the output with respect to each inputs (`grad_w` and `grad_b` fields). Each gradient is then accumulated in its gradient fields and it returns its gradient of weight (`grad_w` field). The structure used to define the `module` class requires 3 methods `forward`, `backward` and `params`, this is used to implement the basic layers; i.e. activation layer.

IV. Code structure

In this section we will describe the content of the files in the library *RocketDeepL*.

- `relu.py`: activation class that implements the ReLU operation (non-linear function).
- `sequential.py`: this class is used to handle different classes. As input it takes a list of layers that compose the neural net.
- `sgd.py`: optimizer class that it is initialized with a *learning step* parameter.
- `tanh.py`: activation class that implements the hyperbolic tangent operation (non-linear function).
- `l_mse.py`: class that implements the mean square error.
- `layers.py`: it implements the class Linear that is a wrapper of two operation: sum of biases and multiplication of weights ($y = xw + b$).
- `module.py`: abstract class *Module* containing the basic methods.

V. Activation layer

We have implemented two activation layers; the first use *reLU* as activation function, the second one *tanH* function. As explained in Section I activation function are essential to guarantee a good accuracy.

- *ReLU* is one of the most used activation function, is defined in an interval $[0, \infty)$. When we must handle with negative values, however, all the values become zero and that cause a bad fit or train of our data. File: `rocket_deepl/core/activations/relu.py`
- *tanH* is defined in the interval $(-1, 1)$ and it is differentiable. It is mainly used for a binary classification. One of its strength is to map negative inputs to strongly negative values and the zero to a value close to zero. File: `rocket_deepl/core/activations/tanh.py`

VI. Learning with gradient descent

Gradient descent is the algorithm that we have used in order to minimize the mean square error (*MSE*). These quadratic function is expressed as:

$$\mathcal{L}(w, b) = \frac{1}{2n} \sum_x ||y(x) - a||^2$$

where w are the weights, b are the biases that we want to find in order to minimize the cost function; n is the total number of training inputs and a is the vector of outputs from when x is the input. In particolar, to speed the learning we have decided to use a *stochastic gradient descent* algorithm. The update of a parameter p, will be expressed as $p = p + \eta \nabla \mathcal{L}(p)$. **??**

The implementation of SGD can be found in the file `rocket_deepl/optimizer/sgd.py`.

VII. Linear layer

The parameters that are trainable in a *fully connected layer* are the weights and biases. However, it is important to notice that these values are initialized from an uniform random distribution, in a range $[-std, std]$, that gives a starting point to our stochastic gradient descendent. Biases are used to compute the output in the other layers. It is expressed as follow,

$$y = xw + b \text{ where } y \in R^{1xD},\ x \in R^{1xM},$$
$$W \in R^{MxD} \text{ and } b \in R^{1xD}$$

All the corresponding implementation can be found in the file `rocket_deepl/core/layers.py`.

VIII. Results

We have tried our deep learning framework with the following specifics (see Fig. 4):

- *inputs*: two nodes
- *outputs*: two nodes (binary classification)
- *layers*: three hidden layers of 25 nodes.

As last we have applied the activation function *TanH* in order to have a bounded output (in range $[-1, 1]$). This make possible to avoid any huge penalization of large positive and negatives values when we use *MSE*. This is crucial in the case we have a binary classification.

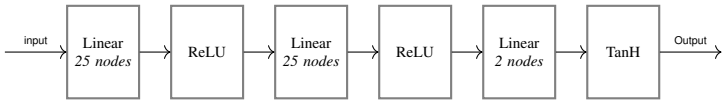

Fig. 4: Structure of model that we have used for testing.

IX. Comparison of Model

TODO: compare with pytorch!!!!!!!

X. Bonus

XI. Result

In this report, we have designed a multi-layer perceptron (MLP) capable using the standard math library and the basic tensor operations of Pytorch. We have trained it using stochastic gradient descent (SGD). The result obtained are really close to the one of the official pytorch *nn library*.