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 to tackle a wide range 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 simple neural networks is composed by an input layer, a series of hidden layers and an output layer (see Fig. 1). Each layer has a certain number of neurons (nodes) each of which processes all the features computed by the previous layer; the nodes of the input layer process the features of the input samples. In a classification task, the number of nodes of the last layer is usually equal to the number of classes to be detected.

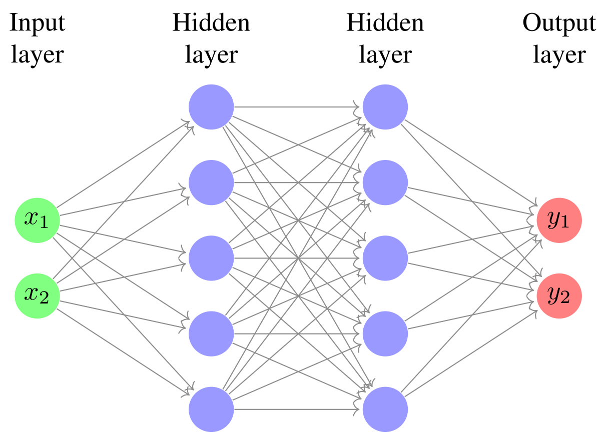

Fig. 1: General neural network with two hidden layers.

Perceptron is the key concept of a neural network, it is a linear classifier (binary) which decides whether or not each input belong o a certain class. Some *parameters* are then introduced so that the obtained parametric function can be optimized in order to obtain the correct output for each input sample. The output of the network is unbounded and is obtained by computing $\sum_i w_i x_i + b$ where w_i is a weight and x_i is an input feature. The *bias* b allows to translate the linear function that would otherwise intersect the origin of the axes. Then, the classification is obtained by comparing this output with respect of a threshold which is usually set to 0. Formally, the output is computed as follows:

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

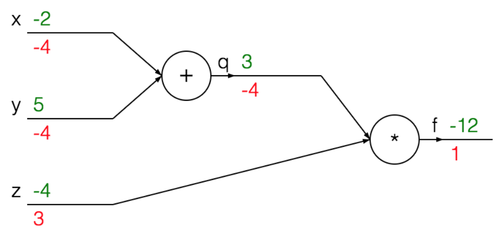

Fig. 2: Example of forward and backward passes. The first is represented in green and the second in red. [**??**]

To make the neural network also learn to discriminate non-linearly separable samples, non-linearities have to be introduced. In particular, *activation layers* (see Sec. V) are used after each linear layer, i.e. the output is obtained by $\sigma(wx + b)$. Since σ is a non-linear function, it allows the network to approximate more complex function and, usually, guarantee a better accuracy.

In our implementation, weights and bias are initialized from a uniform random distribution 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, after this random initialization, we expect our neural network to achieve an accuracy of around 50%. The main core of the project is to define and implement the learning algorithm. First of all, we apply the *forward pass*, then *backward pass* and, finally, we update all the *parameters*.

1) *Forward pass*: computes values from inputs to output.
2) *Backward pass*: start from the computed loss and performs backpropagation by applying recursively the *chain rule* in order to compute all the gradients of the loss with respect of the various parameters of the network. The 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 using the gradients computed during the *backward pass*. In particular, all the parameters are updated in the opposite direction of their corresponding gradient.

In the following sections we will analyze deeply the architecture and we will provide a description of the structure of the code of our framework. Let's 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

The data used for training and testing is 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). The labels 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`.

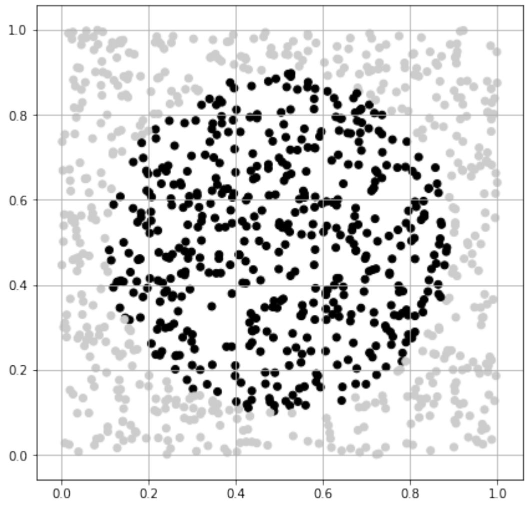

Fig. 3: Points are sampled distributed in range $[0, 1]^2$. Points labelled with 0 are shown in gray, else in black.

III. Implementation

We have decided to use a modular approach in order to make possible to modify or expand it. Backpropagation has been implemented storing all the information of the graph, so that each node keeps track of its input other than calculating its result. Then, to compute the derivative of its input wrt the loss, we can just compute the gradient of the input wrt the result of the node and apply the chain rule to update the gradient obtained by the next operation. The computed gradient is then passed to the inputs. The forward pass just computes 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 the gradient field of the corresponding parameter (`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; e.g. linear and activation layers.

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, the activation functions have been proved to be essential to approximate more complex functions and guarantee a good accuracy.

- *ReLU* is one of the most used activation function, its output is defined in an interval $[0, \infty)$. However, when the input smaller than 0 then also all the gradient are 0s which lead to a null update of the model parameters. This causes a bad train of the model. File: `rocket_deepl/core/activations/relu.py`
- *tanH* its output is defined in the interval $(-1, 1)$. This activation is is differentiable on all R. It is mainly used for a binary classification. It maps negative inputs values close to -1 and positive inputs to values close to 1. 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*). The latter is defined as:

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

where y represents the model which depends on the weights w and biases b that are the parameters that we want to optimize in order to minimize the cost function; n is the total number of training inputs and a is the vector of correct outputs corresponding to x. In particular, to speed the treaining process 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. 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.

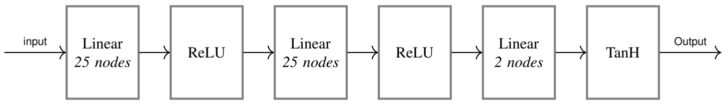

Fig. 4: Structure of model that we have used for testing.

The input and the hiddel layer are followed by a ReLU while the output layer ye followed the *TanH*. This allows to obtain a bounded output (in range $[-1, 1]$) which makes it possible to avoid a huge penalization of large positives and negatives values when using *MSE*. This is crucial in the case we have a binary classification.

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*.