Ecole Polytechnique Fédérale de Lausanne

EE-559 - Deep learning
Mini-projects

Project 2 – Mini deep-learning framework

Nikitas Papadopoulos - 262824

Nicolas Kieffer - 270647

Sarah Fleury - 262329

May 2020

Robotics Master 2 - Summer semester

1. Introduction

The goal for this second project is to design a rudimental Deep Learning framework without using any of the autograd or neural-network module functions. Therefore, these functions will be coded by ourselves only by using pytorch's tensor operations and the standard math library. The neural network will be used for binary classification, as it will try to predict which elements belong in a generated disk, centered at (0.5, 0.5) of radius $\frac{1}{\sqrt{2\pi}}$ in an environment with randomly created points in $[0,1]^2$. If outside of the circle, it has a label 0 and 1 if inside.

2. Theory

The implemented functions that constitute a neural network are the following:
Layer initialization: Random initialization of weights and bias.
Forward: Calculation of the process, gives values of the output layers from the inputs data, by traversing all neurons from first to last layer. The following calculations are conducted with parameters such as weights and bias :

$$x^0 = x, \forall l = 1, ..., L, \begin{cases} s^l = w^l x^{l-1} + b^l \\ x^l = \sigma s^l \end{cases}$$

Backward: Function that updates weights according to the output gradient loss, using gradient descent algorithm, from last layer to first. This is the part where the neural network essentially learns, as it computes the derivatives of the loss with respect to the activations:

$$if\ \ l > L,\ \frac{\partial \ell}{\partial x^l} = (w^{l+1})^T \frac{\partial \ell}{\partial s^{l+1}}$$

And the derivaties of the loss with respect to the parameters:

$$\frac{\partial \ell}{\partial w^l} = \frac{\partial \ell}{\partial s^l}(x^{l-1})^T$$
$$\frac{\partial \ell}{\partial b^l} = \frac{\partial \ell}{\partial s^l}$$

ReLU: Linear activation function, showed on the following graph:

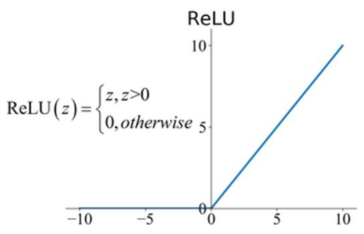

Figure 1: ReLU function

TanH: Nonlinear activation function, showed on the Figure 2:

MSELoss: Mean Squared Error Loss. A commonly used cost function for optimization problems.

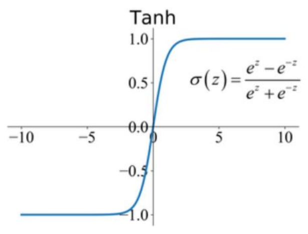

Figure 2: TanH function

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

3. Implementation

To implement the basis of our Neural Network framework, we have separated the classes and the useful functions into two separate files: `module.py` and `function.py`.
In `module.py`, we have implemented the `Module` class with a basic structure: `init, forward, backward and param` , following the principles introduced in the theory part. Each element that wants to build a layer inherits from this base class: `Linear, ReLU, Tanh and MSELoss`. Thus, they all have `init, forward, backward and param` although the activation functions `ReLU`, `Tanh` and `MSELoss` do not have any parameters so the `param` function is empty. The `Linear` class also has one additional function, `update_param` necessary for the weights and bias update, taking as argument the learning rate η. The other essential class implemented in `module.py` is the Sequential class. This latter enables our framework to build a Multi Layer Neural network. To create such an architecture, one can create a model from the class `Sequential` and pass in argument the list of layer with its input and output dimension and the activation layer chosen.

In the `function.py`, we have implemented the useful functions, more general and not belonging precisely to one class. The `convert_to_one_hot_labels` enables to create one-hot-encoding for our model to better learn the classification problem, which in this case is to decide whether the point is inside or outside the circle. A tensor (2X number of points) is created to label the train and test inputs. The function `generate_disc_set` generates train and test sets with their labels with as argument the number of points desired. It contains the `convert_to_one_hot_labels` to obtain directly the label too. The most important function that allows our model to learn is the `train_model`, taking as arguments:

- model, which is the object from class Sequential, it contains all the information about the different layers and activation function
- train_set and train_label, the data created and labeled using one-hot encoding in or out the circle.
- loss, which is an object of MSELoss class.
- mini_batch_size, in which one can choose the size of the mini batch for the stochastic gradient descent.
- eta, the learning rate. one can tune this parameter to improve the training

- n_epochs, is the number of times the whole dataset will go through the neural Network.

This function has two `for` loops, one for iterating over `nb_epochs` and one over the `mini_batchs _size`.
The Stochastic Gradient Descent was directly implemented in the `train_model`. Thanks to the iteration over the `mini_batchs`, the model is fed with partial dataset from the `train_set` of `mini_batch_size`. Then, the loss is calculated to evaluated how far the output of the training is from the label. Then, with the backward step, the function takes as argument the gradient from the loss output and goes through each of the backward functions of each layer to update the gradient of the weight and bias. Finally, weights and bias and updated by going through the parameters of all layer stored with the function `update_param`. This step is repeated by the number of epochs for each `mini_batch`. In this way, the parameters update to minimize the loss and learn the classification presented.
The last function is `err_classification`. It counts the number of errors our model gives by giving as arguments the output of test_set and the test_labels.

4. Results/Performance

The model we had to implement was a Multi layer Neural Network with two input units, two output units and three hidden layers of 25 units. The layers chosen were Linear with ReLU activation function and the last one with Tanh activation function. After fine-tuning the parameters, we chose $\eta = 0.01$, `mini_batch_size`=10. We obtained the following result shown in Figure 3.

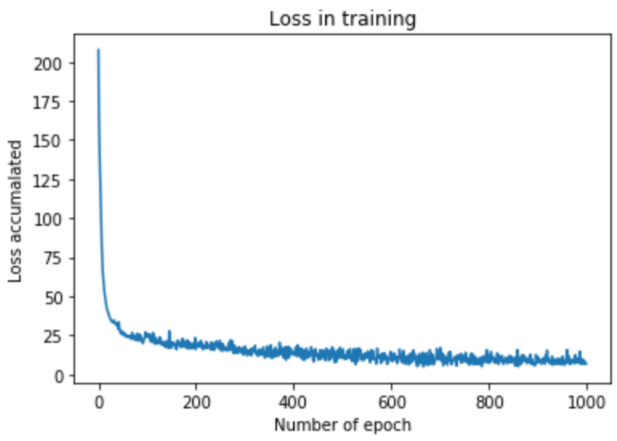

Figure 3: Loss in training set

In our file `test.py`, we displayed the mis-classification obtained for the test_set which is equal to 29 on 1000 points.
These results show that our model learns because the loss over the training decreases until attaining an overfitting plateau. The test error, 2.9%, gave us satisfactory results.

5. Conclusion

The aim of this project was to build a framework to implement a multi layer neural network by performing a forward and backward pass and updating the weights and bias. By going through all these steps, our model can learn to accurately classify points that are found inside or outside the generated disk.