Deep learning framework for 2D point classification problem

Aurélien ORGIAZZI, Gaspard VILLA, Leonard KARSUNKY
EPFL, EE-559, Deep Learning

***Abstract*—In this project, the main objective is to build a deep learning framework using neural networks "from scratch", i.e. without using autograd or the neural network modules already implemented in PyTorch. The idea is to show the structure of a python piece of code for a neural network and see how it can be improved with different classic methods for neural networks.**

I. INTRODUCTION

In this project, we will build a python code base to create a mini deep learning framework, resembling the functionalities of PyTorch with much simpler implementation and capabilities. The problem with which we will test our framework is a classification problem, i.e. in our case to classify points in a circle. Specifically, the points should have a label 0 if outside the disk centered at $(0.5, 0.5)$ of radius $1/\sqrt{2\pi}$ and 1 otherwise. A figure showing these points in $2D$ can be seen in Figure 1. Our executable code `test.py` will build a standard neural network, more precisely a multi-layer perceptron, which will contain two input units (our $2D$ point to classify), one output unit (either 0 or 1), and three hidden layers of 25 units each. Here we will show how our initial simple yet functional architecture is built, then modifications with regards to the activation functions and loss functions will be examined and tested.

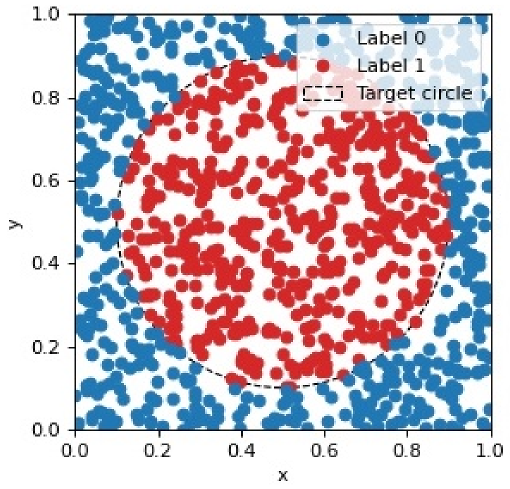

Figure 1: The training set with color for each label plotted with the target circle centered in $0.5, 0.5)$ and with radius $1/\sqrt{2\pi}$.

II. INITIAL FRAMEWORK STRUCTURE

To implement our framework, we extensively used object-oriented programming (OOP). In our initial implementation, we created a parent class called `Module` as a unifying class from which all of our child classes inherit. This class contains the following functions: `forward`, which gets for input a tensor and returns a tensor, and `backward`, which gets as input a tensor containing the gradient of the loss with respect to the module's output, accumulates the gradient w.r.t. the parameters, and returns a tensor containing the gradient of the loss w.r.t. the module's input. It also contains `param`, which represents the layers of our network, for instance `Linear` or `ReLU`. The child classes represent roughly the same classes as in the `torch.nn` PyTorch module, and have the following functionalities:

- `Sequential`: The class by which our model is instantiated. It takes into account all the different layers of our neural network, and does the forward and backward passes on each layer during training.
- `Linear`: Fully-connected layer that applies a linear transformation to its input. It contains the weights and biases as well as initializing them. It also contains our stochastic gradient descent optimization algorithm with a constant learning rate. The `backward` function updates the parameters of the network using the back-propagation algorithm and stochastic gradient descent. This is how our neural network "learns".
- `LossMSE`: Loss function of our model. As part of stochastic gradient descent, the error of the current state of the model must be estimated repeatedly during training. We chose here the mean squared error (MSE) as our loss function which measures the average of the squares of the errors.
- `ReLU`: Rectified linear unit activation function is a nonlinear function allowing complex relationships in the data to be learned, which simply outputs the input directly if positive and zero otherwise. ReLU is a technique that now permits the routine development of very deep neural networks, mostly for MLPs and CNNs.
- `Tanh`: Hyperbolic tangent activation function, this "S-shaped" nonlinear function takes any real value as input and outputs values in the range -1 to 1. However, it is more susceptible to the vanishing gradients problem which is more for very deep networks with

many layers where the gradient slowly becomes zero as we go from output layers to input layers. But, we are not concerned with that issue since the small size of our network.

III. Testing different activation functions

After our first tests on these modules, we noticed that when using ReLU, approximately half of the time the train and test error were near 50%, corresponding to the worst scenario possible. It is shown in Table I some results obtained with different activation functions. After some research, we realized that we faced the problem of "dying ReLU" during training. This problem refers to the scenario when a large number of ReLU units only output values of 0. This becomes a problem when a majority of the inputs from these ReLU units are in the negative range. This can turn off a large part of the network that is no longer able to learn [1]. To solve this problem, we chose to implement several other activation functions to avoid the flat segment of ReLU in the negative range and see which of them gives improvements. Thus, we have implemented the Leaky-ReLU, exponential linear unit (ELU) and sigmoid functions.

To begin with, we implemented the sigmoid function. It is also an "S-shaped" function as Tanh, but the sigmoid has its asymptotes in 0 and 1, instead of -1 and 1. In fact, the activation function Tanh is just a linear combination of a sigmoid function with parameter $\lambda = 1$. This being said, the sigmoid is mathematically defined by the following equation:

$$s(x) = \frac{1}{1+e^{-\lambda x}} \tag{1}$$

For setting the parameter λ, we have tested different values and ultimately chose it to be $\lambda = 3$, because it seems to be the one showing the best results. With this activation function, we have better results than with ReLU, in fact we have at the end 13.69% of error with the test set, with a standard deviation of 13.83%. The results in Table I show that the sigmoid gives better results, but still has some problems to give a good classification of the points.

After that, we implemented the exponential linear unit (ELU) function. The idea for this function is to get another version of the ReLU and instead of setting it to 0 when it is negative, we set it equal to an exponential term. The purpose is to have something smooth at 0 and still keeping an asymptote close to 0 equal here to $-\alpha$ (see the equation (2)). The goal is to prevent the "dying ReLU" issue by avoiding the zero-slope segment in the negative part. The mathematical expression of this function is given by the following equation (2).

$$elu(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(e^x - 1) & \text{otherwise} \end{cases} \tag{2}$$

For this function, α was chosen to be equal to 0.01 after some trials for getting the best one. Clearly, with this activation function, we have the expected improvements with an error of 5.19% on the test set with a standard deviation of 1.37% (see Table I). This means that the results are stable at a low level of error, implying that the architecture is well adapted for this kind of problem.

Then, we implemented the Leaky-ReLU rectifier. This activation function is very close to ELU. In fact, instead of having an exponential saturation on the negative range, it adds a slight slope of coefficient β. Like ELU, the goal is to prevent the "dying ReLU" issue by avoiding the zero-slope segment in the negative part. Thus, the function is defined as follows:

$$leaky(x) = \max(\beta x, x), \tag{3}$$

with $0 \leq \beta < 1$ very small. For β, we chose the value 0.01. The results for Leaky-ReLU are also very promising; indeed we got 4.45% on average for the test error with a small standard deviation of 1.08%. The results are even better compared to ELU, but still very close.

This being said, we will specify how different activation functions were tested with our architecture. Each activation function will be tested over 20 rounds (i.e. 20 identical models with different initialization) that includes 20 different test sets, each containing 1000 samples, to make sure that the mean error represents a good average of the efficiency of the model. Then, the models will be trained over 100 epochs, with a mini-batch size of 10. All the train and test errors of the different activation functions of the network are summarized in Table I that follows with their respective standard deviation.

Activation function	Train error [%]	Test error [%]
`ReLU`	30.36	33.15 ±22.11
`Sigmoid`	12.08	13.69 ±13.83
`ELU`	4.13	5.19 ±1.37
`Leaky-ReLU`	3.62	4.45 ±1.08
`Tanh`	3.59	4.43 ±1.28

Table I: Train and test errors for each activation function using MSE loss, with their corresponding standard deviation.

The general interpretation of these results was already done before, but we will give a small summary of that. So, due to the dying ReLU problem explained previously, we were looking for some other activation functions that are more stable for our problem. Looking at the results, it seems clear that Leaky-ReLU and Tanh are the best candidates for our architecture. By arbitrary choice, we chose Tanh for testing the other features of our architecture.

IV. Additional features

In this section, we will present the main improvements that can be added to the architecture and see how we can implement them into it. This can be done in many ways, but for our project we did this in three steps: firstly, adding an Adam method for learning rate decay; secondly, using Cross-Entropy loss instead of MSE; and finally, adding dropout in our network for training.

First, we decided to implement the Adam (Adaptive Moment Estimation) gradient descent optimization algorithm to improve the optimization and get better results more efficiently. For this, we added a parameter `lr-method` in classes `Sequential` and `Linear` in order to choose if we want the classic method with constant learning rate or an Adam method for the learning rate decay. The parameters for Adam method were chosen such that the gradient descent is the most efficient (see in the class `Linear`).

Then, we implemented the Cross-Entropy loss function in order to compare the results with MSE loss. Indeed, this loss function is more suitable for a classification problem, because we want to heavily penalize cases where the network is predicting the wrong output class. Therefore, it seemed interesting for us to implement it and we tested it with Tanh to compare it with MSE loss. In Table II, the results show that Cross-Entropy loss doesn't improve the classification, it seems to be quite the same. It must be said that the results were already very good using MSE loss.

Finally, we created a dropout module to see if our results could be improved with this functionality. To set this up, we created a class `Dropout`. The principle of dropout consists of a forward pass, where some units are deactivated randomly with a probability p (dropout rate) and scale the units by $1/(1-p)$. We chose to set the backward pass as just returning the gradient (input) without changing it. Then, we created both an `Eval()` and a `Train()` mode in the class Sequential to activate or not the dropout functionality, depending on whether or not it is in the training or testing phase. To test this dropout module, we used the activation function Tanh with both MSE and Cross-Entropy losses.

As we can see in Table II, adding this feature failed to improve results. Nonetheless the results are not bad and stay close to the previous results for Tanh. A possible explanation is that Dropout is especially effective for very deep networks, where it is important to ensure independence between units. In our case, the network is quite small, so Dropout is not very useful.

Models	Train error [%]	Test error [%]
`Cross-Entropy loss`	4.27	4.74 ±1.30
`MSE Loss and dropout`	4.38	4.49 ±0.85
`Cross-Entropy Loss and dropout`	5.75	6.39 ±1.13

Table II: Train and test errors for Tanh with the corresponding standard deviation, using dropout (dropout rate of 0.2)

V. A final plot

To finish this project, we were curious to see what the classification of our model would look like and see to what degree it is close to the target circle. The final plot is represented in Figure 2 and was obtained with the Leaky-ReLU activation function. Since cross-entropy loss was used for training, it is interesting to see a kind of probability of how the model is sure about the classification of the points.

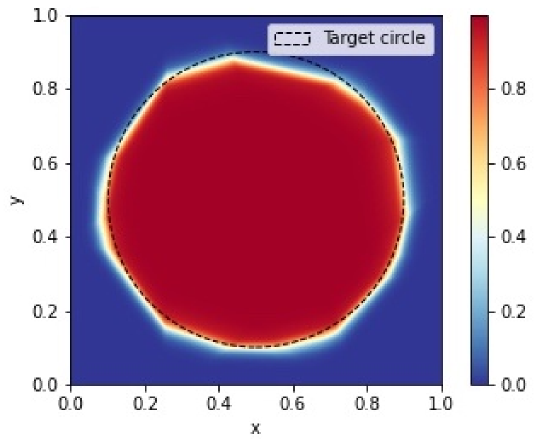

Figure 2: Classification of 2D points in a circle centered in $(0.5, 0.5)$ of radius $1/\sqrt{2\pi}$ given by a trained neural network with activation function Tanh and Cross Entropy as loss function.

VI. Conclusion

Thus, in this project, we have designed a mini deep learning framework, with the main neural network modules of PyTorch such as Tanh, MSE, Linear and Sequential. This allowed us to better understand all the mechanisms of a neural network, in particular for the forward and backward passes. Furthermore, we made several tests to improve our model by implementing other modules or functionalities, such as different activation functions, the Adam optimizer, cross-entropy loss and dropout.

References

[1] Lu Lu, Yeonjong Shin, Yanhui Su, and George Em Karniadakis. Dying relu and initialization: Theory and numerical examples. *arXiv preprint arXiv:1903.06733*, 2019.