Deep Learning Project 2 - report

Maïk Guihard (284922), Alexandre Clivaz (287897), Bruno Da Costa (288525)

May 2021

Introduction

For this mini-project, the goal was to develop a mini-framework resembling torch. The modules from this framework should be usable to classify a set of points in $\mathbb{R}^2$ using a multi-layer neural network.

Training & testing data generation

In order to train and test our framework, a function was created to generate a dataset of X samples. Each sample is a tensor of size two, uniformly distributed in [0,1]. Then, a label is assigned to the data. If the euclidian distance (in $\mathbb{R}^2$) from the point to (0.5, 0.5) is smaller than $1/\sqrt{2 * \pi}$, the label is 1, otherwise it is 0. The training and testing datasets are generated the same way. As we are mainly interested in classifying the points in [0,1]x[0,1], having a uniform distribution in this interval is important.

Framework Implementation

In order to implement our mini framework, we decided to follow the indication from the project description and create a **Module** class, from which all our modules inherit. The four default methods all modules have are **__init__**, **forward**, **backward**, and **param**. In the **Module** class, these four methods simply raise the NotImplementedError, this way if one of our modules is not fully implemented, this error pops up and we can implement the missing part.
The modules that were implemented are the following:

- Sequential
- Linear
- ReLu
- leaky_ReLU
- Tanh
- Sigmoid

And the loss used is the MSELoss. Some modules have internal parameters, stored internally in tensors, as well as other tensors that accumulate these parameters' gradient. The backward method of each module stores the current input value, so that the loss can be propagated with respect to it in the backwards pass. The particularities of each module are presented thereafter.

Sequential

The sequential module takes as input, when created, a sequence of transformations. The method **forward** simply executes the forward pass of each input module sequentially. Similarly, the **backward** method executes the backward pass methods of each module, but in a reverse order. The **param** method also returns a list of all parameters of each input module.
There are two unique methods in this module: **step** and **zero_grad**. The first one adds the gradient accumulated in each module to its parameters, and the second one resets these gradients to 0. As each module accumulates and stores its gradients internally, the Sequential module doesn't need to store them.

Linear

Upon creation, the linear module creates a tensor of weights and biases, as well as the corresponding tensors for the gradients. These are the parameters of the layer, and can be updated after/during the backward pass respectively. These four tensors are the output of the **param** method.

ReLU, leaky_ReLU, Tanh, Sigmoid

These four modules do not have any internal parameters, therefore their **param** methods return an empty list. At first, only Tanh and ReLu were implemented. We decided to add leaky_ReLU because the gradient was vanishing too fast when backpropagating. Also, with the testing dataset, the expected output is either 0 or 1. Therefore we thought the classification error could be improved by implementing a sigmoid module, and indeed it improved the results by a lot. With ReLU, leaky_ReLU and Tanh, the loss had high-frequency oscillations, even before reaching the optimum, which is not the case with Sigmoid.

Results

In order to test the framework, 1000 train and test samples are generated according to the project description's requirements. The train samples were fed through a network composed of two input units, one output, and three hidden layers of size 25. There is a leaky_ReLU between each layer, and a Sigmoid function at the output unit. The model is trained with the implemented MSE loss, while logging it. To evaluate how well it performs, the classification error rate on the training **and testing** samples is logged. The three graphs are shown on Fig. 1 for a network trained for 400 iterations. The results are satisfying. The trained model has a classification error rate of 3.4% on the train data and 3.2% on the test data, and the MSEloss decreases as expected. We see that the loss continues decreasing, even after the classification error stabilizes. As the classification takes the closes value in $\{0,1\}$, the classification can be right even if the loss is still consequent (for example: the prediction is 0.52, the target is 1: the MSEloss is $0.48^2 = 0.23$, but the classification is already 1). Therefore it appears that the 400 iterations necessary for the loss to stabilize, are not required to reach peak performance in classification. This is partly due to the fact that the problem is fairly simple, with the target being either 0 or 1.

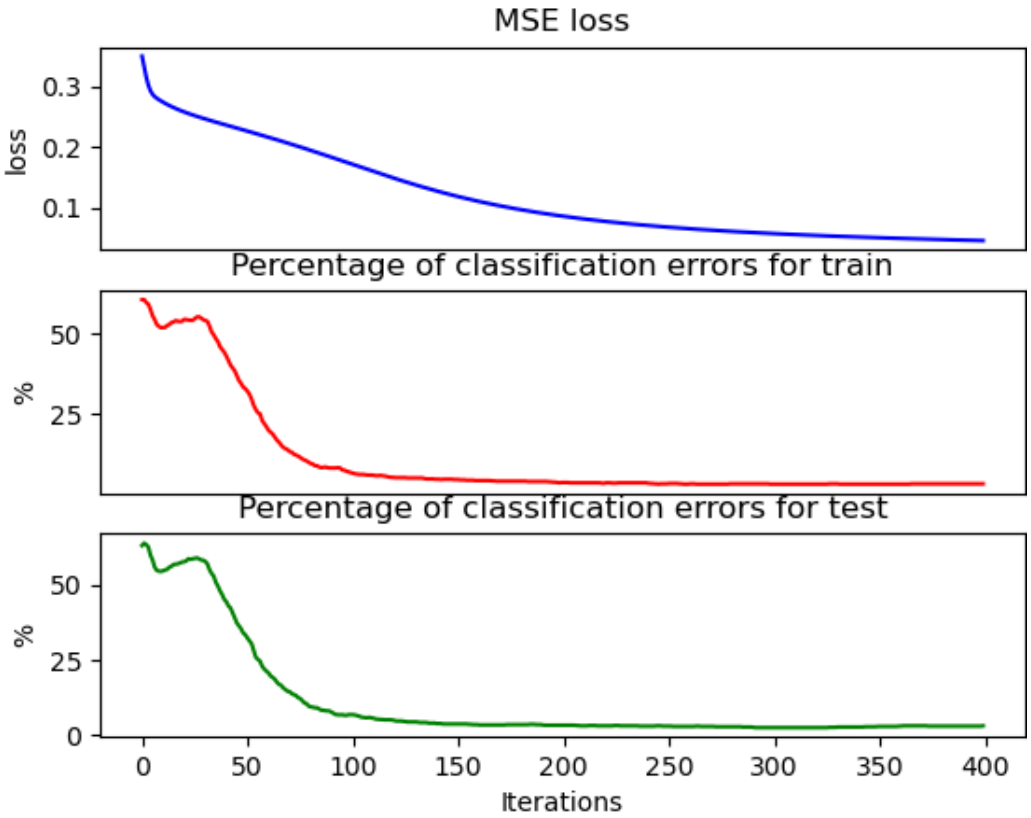

Figure 1: Log of MSEloss, train & test error rate at each iteration

In order to visualize how our model partitions the space, a grid of 100x100 points centered on (0.5,0.5) was created and run through our trained model. The output is shown on Fig. 2, and it corresponds to the way labels are generated: 1 in a circle centered in (0.5,0.5) of radius $1/\sqrt{2\pi}$, and 0 outside. The circle is not perfect for two reasons: the model does not reach 0% of errors, which indicates that the space is not perfectly partitioned, even on the limited number of points that were used for training. The second reason is that the distribution of the 1000 training points in the region is random, therefore some parts of the space may have less reference points than others.

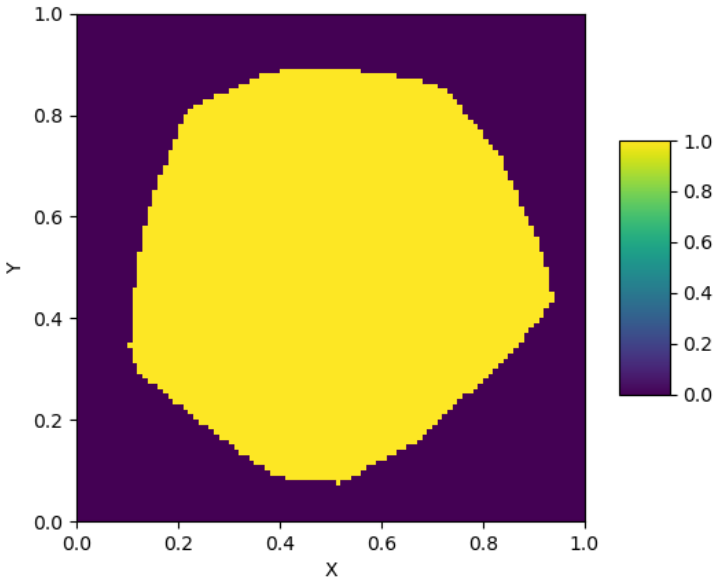

Figure 2: Partition of space by the model