Deep Learning
End of semester mini-projects

Lilian Rouzaire (Group W)

December 17, 2021

For this end of semester assignment, I chose to work on the two-mini projects described on the course page. This document contains the report of both mini-projects and comes along with the Python files. I worked alone on this assignment.

1 Project 1 - Classification, weight sharing, auxiliary losses

1.1 Introduction

Objectives. The aim of the project is to compare two digits on two-channel images with Deep Learning techniques, and to highlight the role of weight sharing and auxiliary losses in this classification problem.

Plan. Given as input tensors of size $2 \times 14 \times 14$ corresponding at images of digits, one will implement neural networks predicting whether the first digit is lesser or equal to the second. The objective is to choose different model architectures and compare the results given by each one of them.

1.2 Chosen model architectures

1.2.1 Weight sharing networks

One will start with basic networks with weight sharing. Two `Module`s are implementing this structure, a `SimpleNetwork` and a `WSNetwork`. The `SimpleNetwork` consists in one convolutionnal layer followed by a ReLU activation function and a linear layer. The `WSNetwork` starts with three convolutionnal layers, then adds two linear layers and ends with a dropout. The activation function is ReLU for all layers, and a 2D pooling is also applied on the convolutionnal layers.

1.2.2 Networks without weight sharing

In the previous modules, the module answer was a binary response ; whether the first digit is lesser or equal to the second. One will now take advantage of the availability of the classes of the two digits for each pair of digits. The chosen strategy is to run a deep model on each digit image separately, and then compare the predictions, so that the weights are not shared during the forward and backward passes. Such networks will therefore use auxiliary loss functions to evaluate the efficiency of the models.

`ALNetwork` implements this protocol. A `MNISTClassificationBlock` module is in charge of classifying the images separatly, using the provided classes. The network is then able to decide whether the first digit is lesser or equal to the second.

While a standard loss function was used for the weight sharing networks (mostly `BCEWithoutLogitsLoss`), running a model without weight sharing requires a auxilliary loss function.

I could not work out a stable custom loss function. Therefore, I could not setup the `ALNetwork` or any other deep learning module working with an auxilliary loss.

1.3 Architectures comparison and discussion

As one did not implement any custom loss function, one can only assess the efficiency of the weight sharing networks. All the simulations are lead over 25 epochs, with batches of size 100 and $\eta = 2 \times 10^{-3}$.

Figure 1: Error rates of two weight sharing networks

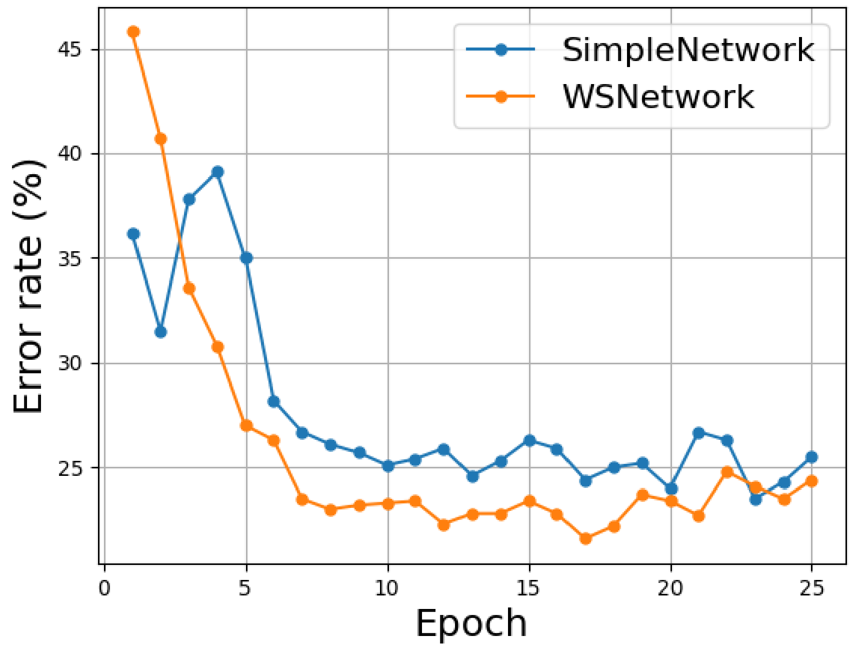


Figure 1 shows the efficiency of the `WSNetwork` against the `SimpleNetwork`, with a dropout probability of zero. The error rate is still quite high after 25 epochs, but the more simple network is slightly less efficient than the one with 3 convolutionnal layers and 2D pooling.

1.4 Conclusion

The only tested architectures were the ones implementing weight sharing. One could not test models with auxiliary loss, but one could assess the performance of a more sophisticated network such that `WSNetwork` compared to a basic network with two layers.

2 Project 2 - Mini deep-learning framework

2.1 Introduction

Objectives. The goal of this mini-project is to implement a small deep learning framework from scratch without using all core functions of PyTorch such as autograd.

Case study. In order to train and test the mini-framework, one shall reconsider the example given in the course materials of the sphere in a cube. More precisely, given a $[0, 1] \times [0, 1]$ cube, and a $C((\frac{1}{2}, \frac{1}{2}), \frac{1}{2\sqrt{\pi}})$ 2D circle, the neural network must classify uniformly-distributed points in the square depending whether they belong to the circle or not.

2.2 Chosen structure

The chosen structure follows the suggested one. Firstly, a abstract `Module` class is created, and defines the methods `forward(self, *input)`, `backward(self, *gradwrtoutput)`, `param(self)` and `zero_grad(self)`, respectively for the forward pass, backward pass, parameter listing, and gradient reinitialization. Naturally, the forward pass needs an input, and the backward pass uses the previous gradient to comply with the chain rule and back-propagation mechanics.

In the implemented network, every block will be a module (and thus inherit from the base class `Module`), which are described in the following subsection. Modules are then stacked up by the network. All the modules are defined in *module.py*, while *test.py* contains the network definition, training and testing.

2.3 Modules list

Linear module. The base component for the forward pass is a `Linear` module. This module applies a linear transformation to the input x to obtain an output s, using as parameters a weight tensor w and a bias tensor b such that

$$s^{(l)} = w^{(l)}x^{(l-1)} + b^{(l)}$$

where the exponent l denotes the layer. The biases are initialized to zero. One provides two procedures for the weights initialization : the "Xavier initialization", and a uniform repartition. Both are described in the course handouts and their efficiency are discussed in the last section. In addition, the `Linear` module keeps track of the gradient with respect to the parameters.

Activation functions. Two different activation functions are provided in the implementation. The first one is tanh, and the other one is ReLU. The former is the usual hyperbolic tangent, whose derivative $\frac{1}{\cosh^2 x}$ is used for the backward pass. The latter is defined by $\max(0, x)$ therefore its derivative is 1 for all $x > 0$, 0 for all $x < 0$ and arbitrarily set at 0 for $x = 0$.

Loss function. The network will be used to categorize points depending on their position, therefore the studied space has a metric and the quantities are continuous. Thus an suitable loss function for this problem is based on the mean square error. The `LossMSE` module stores the target on initialization, and computes the sum of the squared difference between the target and the given output. As seen in a practical series earlier in the semester, the backward pass of such a function is given by $2 \times (\text{output} - \text{target})$. The backward method will be used as an entry point to the back-propagation mechanism, by optimizing the weights and biases of previous layers in order to minimize the loss quantity.

Sequential module. As in all deep networks, one will use several learning units. To this end, it is more practical to group the layers in sequences, so that one can define the forward and backward passes onto the sequence directly, without managing each one of the layers. That is the point of the `Sequential` module, which contains a sequence of submodules. Its forward and backward passes consist in iterating on the submodules forward and backward passes. It also comes with the `param()` method to concatenate all the submodules parameters.

2.4 Network functionning

Dataset generation. A `generate_dataset()` function is provided to create points uniformly distributed in the [0, 1] x [0, 1] square, and compute whether they belong or not to the circle centered in $(\frac{1}{2}, \frac{1}{2})$ of radius $\frac{1}{2\sqrt{\pi}}$.

Network module. The network is itself a `Module`, taking as parameter the input dimension (2 in our case, corresponding to the cartesian coordinates of each point), the output dimension (1 in our case, corresponding to the classification of the point) and the number of hidden units in the layers (which is imposed to 25 in the assignment statement). The module is composed of one input unit, 3 middle layers of 25 units, and one output unit, all linear. The network gathers all these linear layers along with the ReLU activation functions in a `Sequential` structure described in the previous subsection.

Network training and testing. The training set is divided into batches of smaller size, so that the gradient computation does not consider the whole training tensor for each epoch. In each epoch, the model output is computed using the forward pass. The `LossMSE` module assesses the perfomance of the network and is given to the backward pass which optimizes the weights and biases that will minimize the loss. A `print_epoch_results()` helper function is used to display the error rate over the successive epochs. The errors are counted by the `compute_nb_errors()` function, using the closest prediction for each point of the input set as a final decision, and comparing it to the target tensor, which contains the true answers.

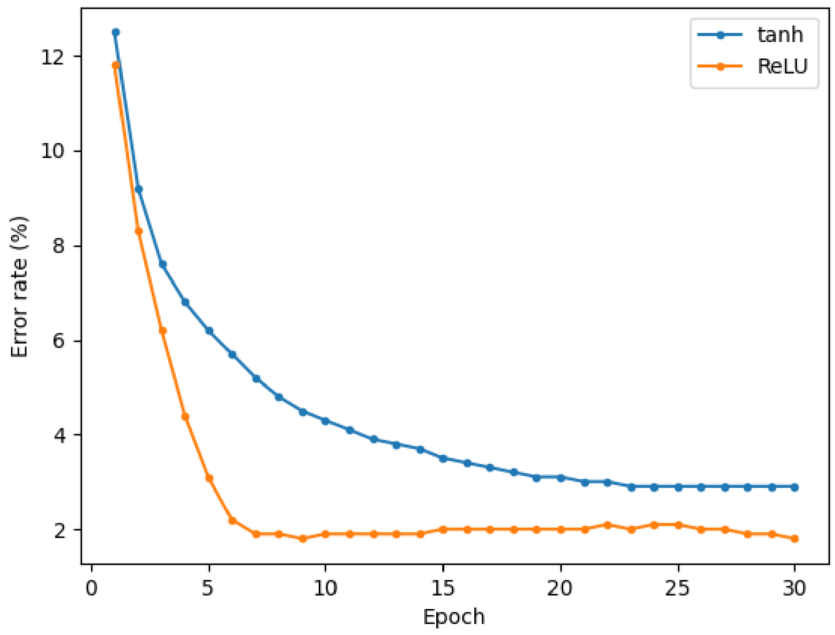
2.5 Results and performance of the network

All data used for the graphs have been used by running the network with $nb = 10000$ data points, a learning rate of $\eta = 0.001$, mini batches of size 10, 25 hidden layers and 30 epochs.

2.5.1 Activation function

Having implemented two activation functions, one might wonder which one of `tanh` or `ReLU` is more efficient in the network. Figure 2 shows the evolution of the error rates for both activation functions. ReLU tends to have slightly better results after 30 epochs, but converges way more rapidly towards a low error rate.

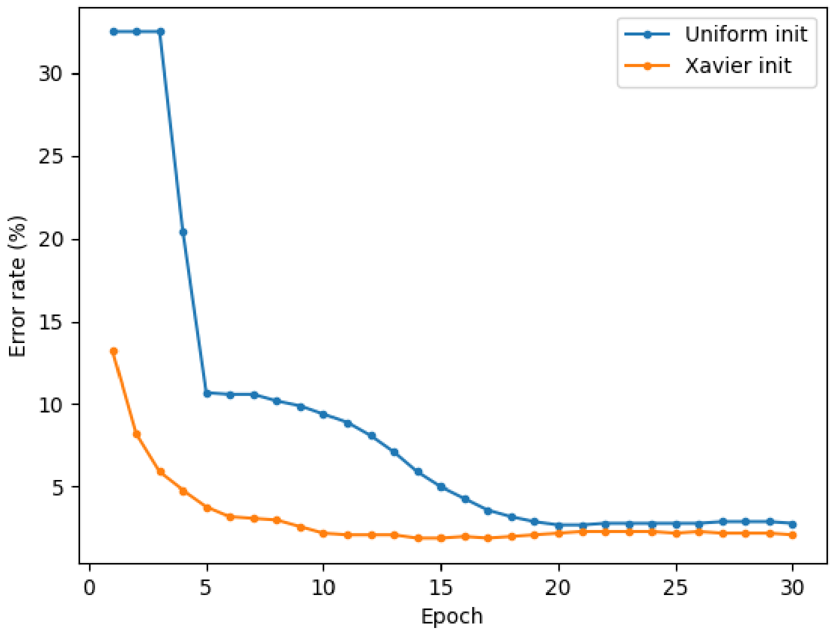
Figure 2: Error rates of two different activation functions



2.5.2 Weight initialization

In the previous sections one described two weights initialization methods. The easiest one is a uniform distribution of the weights, between $\frac{-1}{\sqrt{N}}$ and $\frac{1}{\sqrt{N}}$ (where N is the number of input layers). A more sophisticated one is the Xavier initialization, known to be more efficient. Figure 3 shows how well each one of them perform in the network :

Figure 3: Error rates using two different weight initialization methods

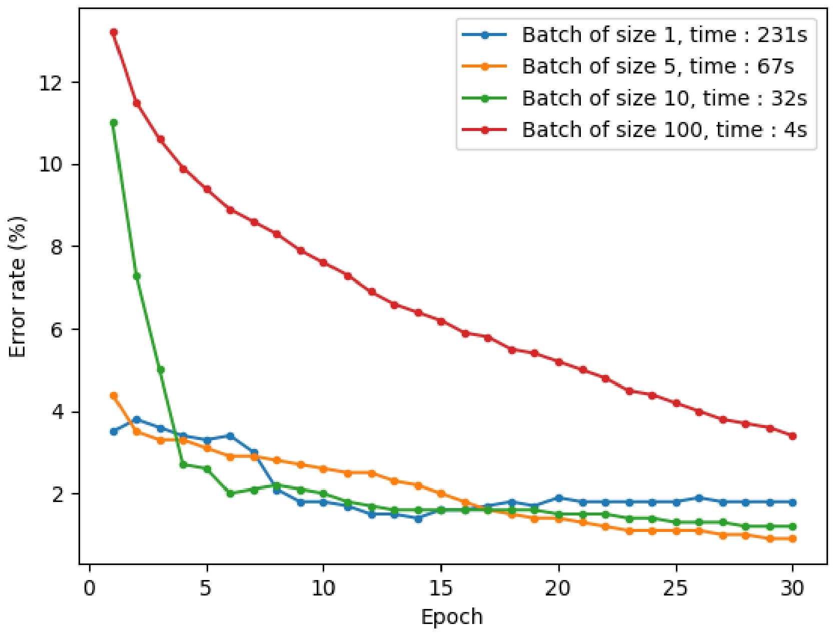


As expected, the uniform initialization is way more unstable in the first epochs, where the Xavier method is way more efficient. In the uniform mode, other simulations always tend to highlight the presence of a plateau followed by a sharp fall for low epochs. At higher epochs, however, both methods are as efficient.

2.5.3 Mini-batch size

One can discuss the impact of the variation of the mini-batch size. Grouping the input dataset into bigger batches provides an important speedup, but figure 4 shows that big batches are very unefficient. For very small batch sizes, however, the reduction of the batch size may not be worth the additionnal computation time.

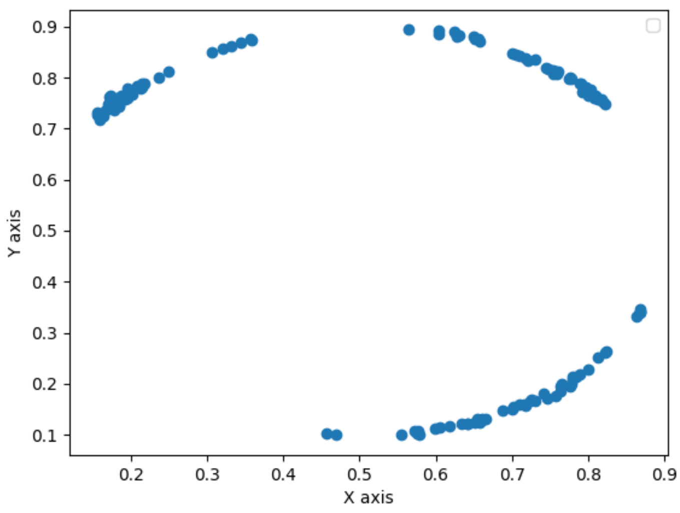
Figure 4: Impact of the batch size



2.5.4 Error repartition

All the previous experiments have been run over 30 epochs, and the average error rate is in the end around 1-5%, and may be influenced by the parameters choice. As in the course materials, one could study the geometrical repartition of the errors after the last epoch. Figure 5 shows the cartesian coordinates of the misclassified points after 30 epochs, with $\eta = 5 \times 10^{-3}$, batches of size 10, 10000 points and 25 hidden layers.

Figure 5: Error repartition



As exepected, the misclassified points are located over a circle of center $\left(\frac{1}{2}, \frac{1}{2}\right)$ and of radius $\frac{1}{\sqrt{2\pi}}$.

2.6 Conclusion

In this project, one has implemented a mini-deep learning framework from scratch, building nested modules and using them in sequences to train a network over a simple classification problem. One has studied the effect of slight changes in the parameters, such as different activation functions, other type of weight initialization or batches size. Finally, one has visualized than the error is distributed over the borders of the circle.