Christopher Straub
Léo Vigna

17.12.2021

Deep Learning Projects

Project 1

The code is divided in two files. The first one “Project1.py” has the training loops for the different architectures. The second file “comparisonNets.py” contains the three architectures.

In order to launch the code you simple run the main script “Project1.py” and it outputs the result of the training and testing of the three architectures in the console.

The three architectures are composed as follow :

1) The input tensor is seen has a 2 channels image. There is a feature extraction with two convolutionals layers and then a fully connected head that predict if the first digit is lesser or equal to the second digit.
2) Each channel of the input tensor is processed by the same network (weight sharing) which try to identify the number with a softmax function with 10 output. Then a fully connected layer try to predict if the first digit is lesser or equal to the second digit.
3) Same as 2, but we use auxiliary loss to force the network to optimize the digit recognition part.

Before each training, we call a function “initParameter” that initialize the weights according to the following rule :

$$f : \mathbb{R}^N \to \mathbb{R}^M$$

$$w_{i,j} \sim \mathcal{U}\left[-\frac{1}{\sqrt{N}}, \frac{1}{\sqrt{N}}\right]$$

Figure 1: Xavier initialization method

Finally, we get those results :

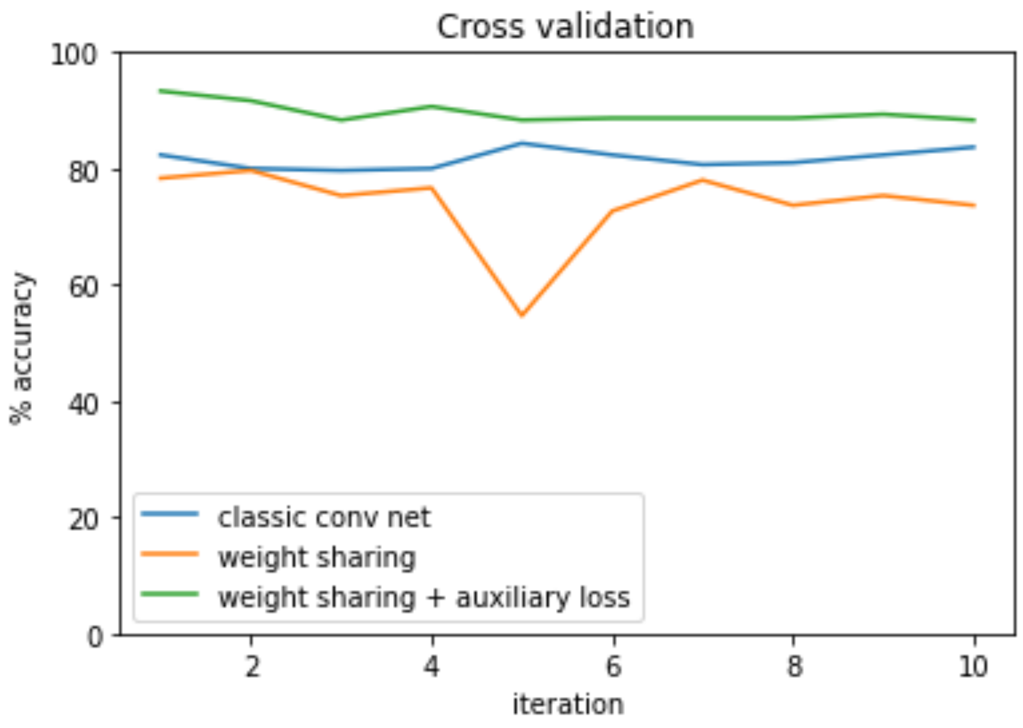


Figure 2: Results cross-validation

We see that the weight sharing and the auxiliary loss outperform the two other architectures. It might be because we ensure that the network learn to recognize a number and then compare the intermediate result to reach to a conclusion. In the two other cases we simply try to optimize a model in order to minimize the binary cross entropy loss of the final decision which seems to be capped around 80% with the given dataset.

Performance with test set	**Accuracy %**
Classic conv net	**80.59**
Weight sharing	**74.59**
Weight sharing + auxiliary loss	**91.00**

A future work could be to tune the loss in order to give a bigger importance to the recognition of the digit first and then optimize the loss of the final decision.

Project 2

There is two files test.py and NNmodule.py. The test.py file uses NNmodules.py to build a neural network, train and test it on a set of labeled points in a square of the plane.

Assuming that a neuron network always consists of compositions of functions, which successively associate each of the linear combinations of its inputs, the output of a continuous function (activation), NNmodules.py creates a neural network using the

Sequential module which takes Parameter modules (Linear) or Activation module (ReLU, Tanh) as argument (Figure 3).

During network initialization, Fields “sigma” and “sigma” are assigned in the activations module with the functions which represent the activation function and his derivative. Finally the class Net is created containing the layers of the network, as well as all the mechanics necessary for its training and its operation.

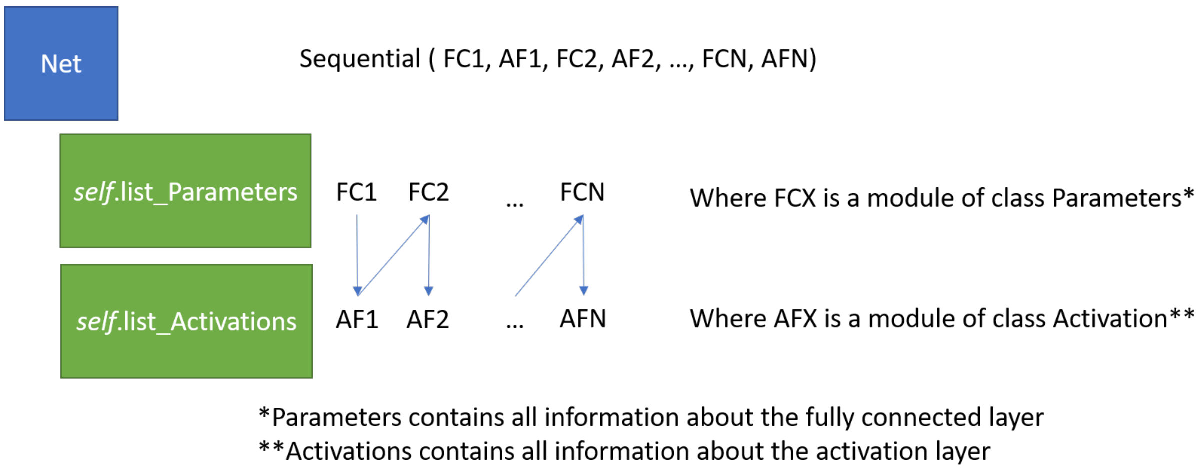

Net
Sequential (FC1, AF1, FC2, AF2, …, FCN, AFN)
self.list_Parameters
self.list_Activations
FC1 FC2 … FCN
AF1 AF2 … AFN
Where FCX is a module of class Parameters*
Where AFX is a module of class Activation**
*Parameters contains all information about the fully connected layer
**Activations contains all information about the activation layer

Figure 3: Structure overview

Each Parameter and Activation module stores in memory the data necessary for backpropagation during the forward. The only values to remember are in the Activation class:

- ds_da: activation function's derivative evaluated in s.
- da_dw: linear function's derivative with respect of this weights (directly the input of the activation function: x (previous layer)).

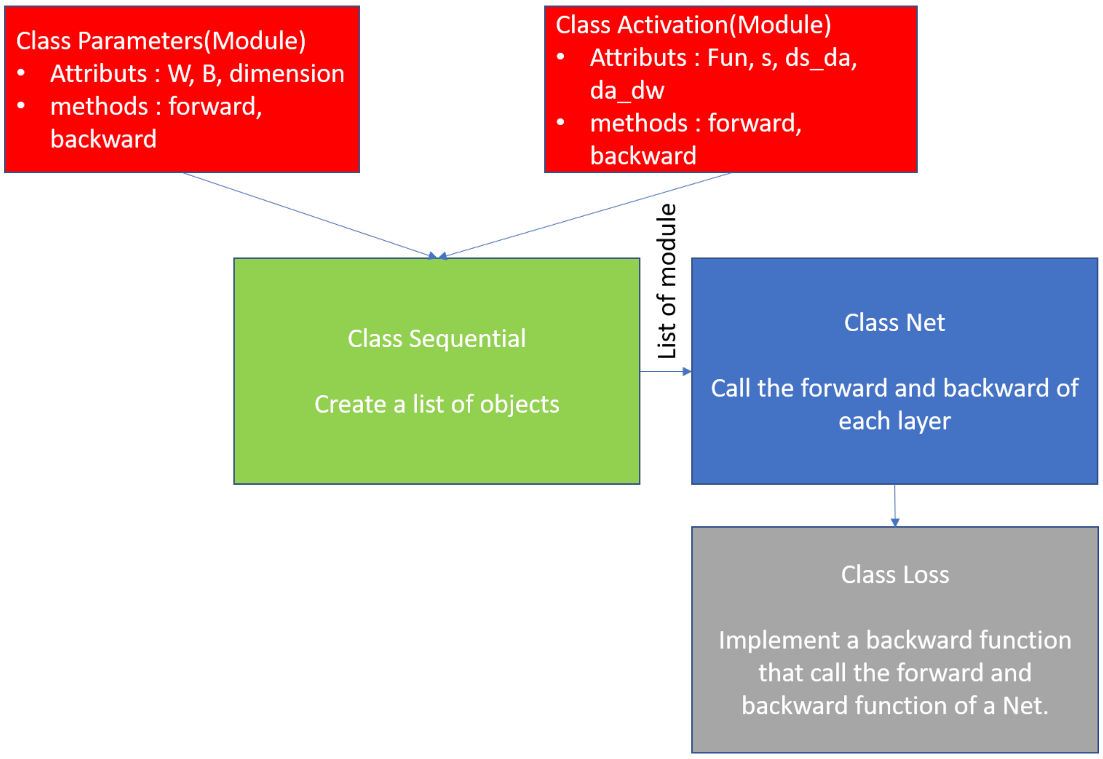

Class Parameters(Module)
• Attributs : W, B, dimension
• methods : forward, backward
Class Activation(Module)
• Attributs : Fun, s, ds_da, da_dw
• methods : forward, backward
Class Sequential
Create a list of objects
List of module
Class Net
Call the forward and backward of each layer
Class Loss
Implement a backward function that call the forward and backward function of a Net.

Figure 4: Global structure

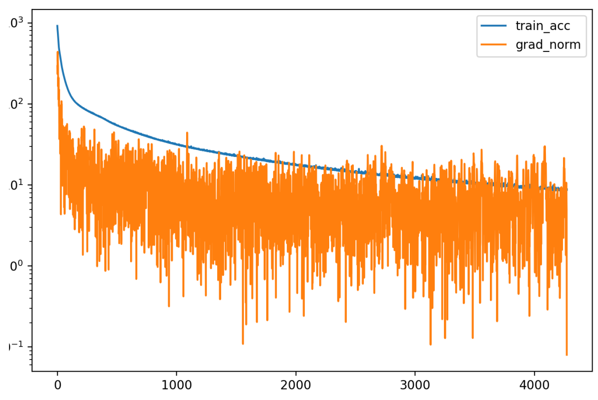
The backward function from Net's module takes as input the loss derivative evaluated at the output of his forward. Like the way forward is implemented, the backward functions of the elements in the Parameter and Activations fields are successively calls in reverse order (right to left).

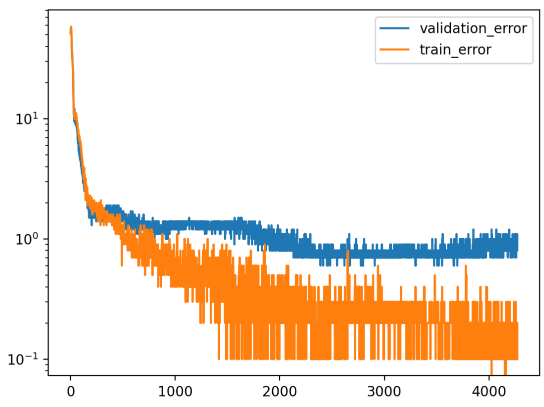
Using the stored values (in Activation), the loss gradients in respect of the linear combinations coefficients from the linear previous layer can be returned by the backward function of the elements of the Net Activation field.

That way the gradients in the Net class can be accumulated, to update the network's weights (which are stored in each Parameter's module) for each new mini-batch with SGD module.

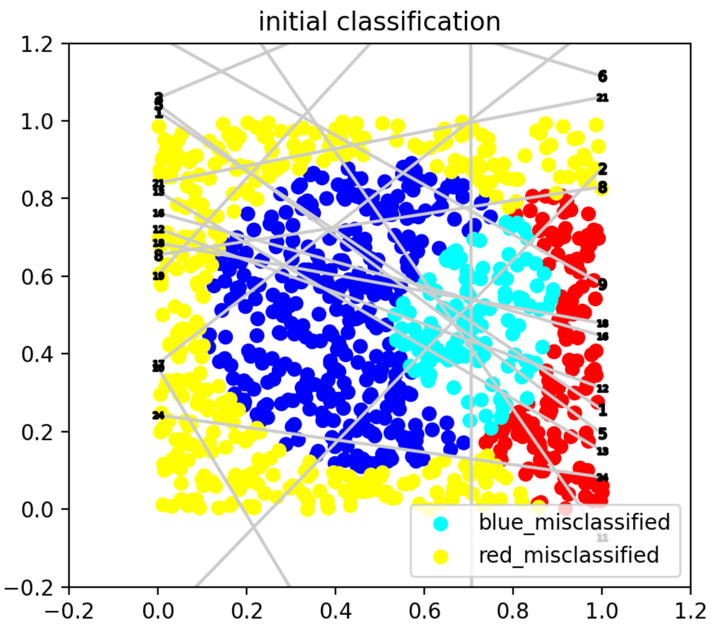
The last two figures represent the results after 4270 epoch for 49.04 seconds training time. The errors rates are : 0.1 % for the training set, 1.0% for the validation set and 1.0% on the test set. The stop criterion in this case was the gradient magnitude which value was 0.08(<0.1).

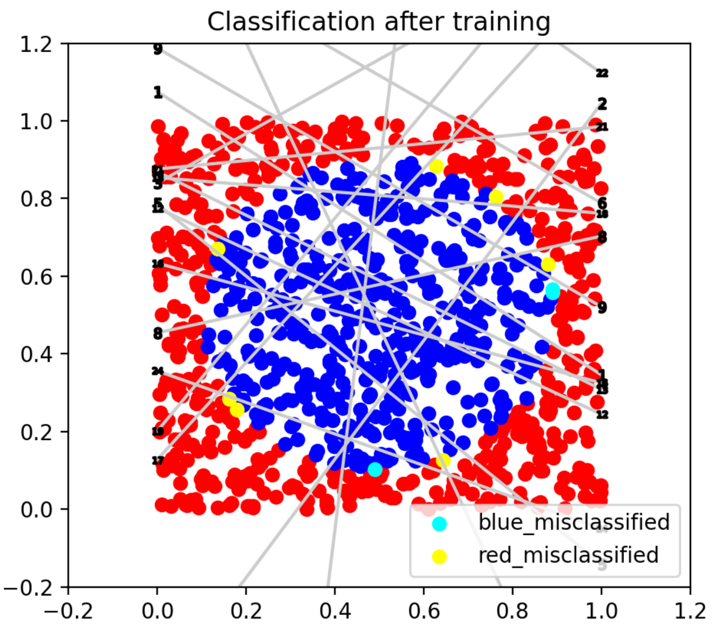
The stop criterion is useful when the initializations weights are poorly conditioned such that the gardien descent is stuck in a local minimum. In that case the stop criterion prevents the training to go through all the 20’000 epochs (in this case, there is 1000 epochs in the submitted files).





The last one represents the test point after classification with the first layer hyperplan for each linear combinaison.


initial classification
blue_misclassified
red_misclassified


Classification after training
blue_misclassified
red_misclassified