Christopher Straub 17.12.2021

Léo Vigna

# 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 :

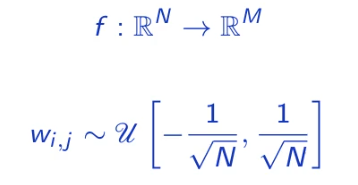


Figure 1: Xavier initialization method

Finally, we get those results :

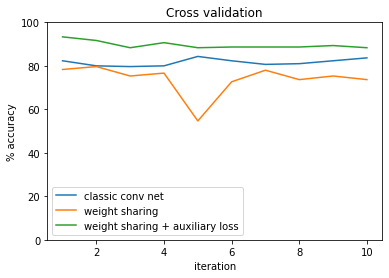


Figure 2: Results cross-validation

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

Le code est divisé en deux fichiers test.py et NNmodule.py. Le fichier test.py utilise NNmodules.py pour construire un réseau de neurone, l’entrainer et le tester sur un ensemble de point labellisé dans un carré du plan.

En supposant qu’un réseau de neurone est toujours constitué de compositions de fonctions, qui associent, successivement, à chacune des combinaisons linéaires de ses entrées, la sortie d’une fonction continue(activation), NNmodules.py crée un réseau de neurone à l’aide de la module Sequential qui prend en argument des modules de Paramètres (Linear) ou module Activation (ReLU, Tanh) (Figure 3).

Lors de l’initialisation du réseau, les fonctions d’activations contiennent des fonctions « sigma » et « dsigma » permettant le calcul des activations et des dérivées de celle-ci. Finalement la class Net est créé contenant les couches du réseau, ainsi que toute la mécanique nécessaire à son entrainement et son exploitation.

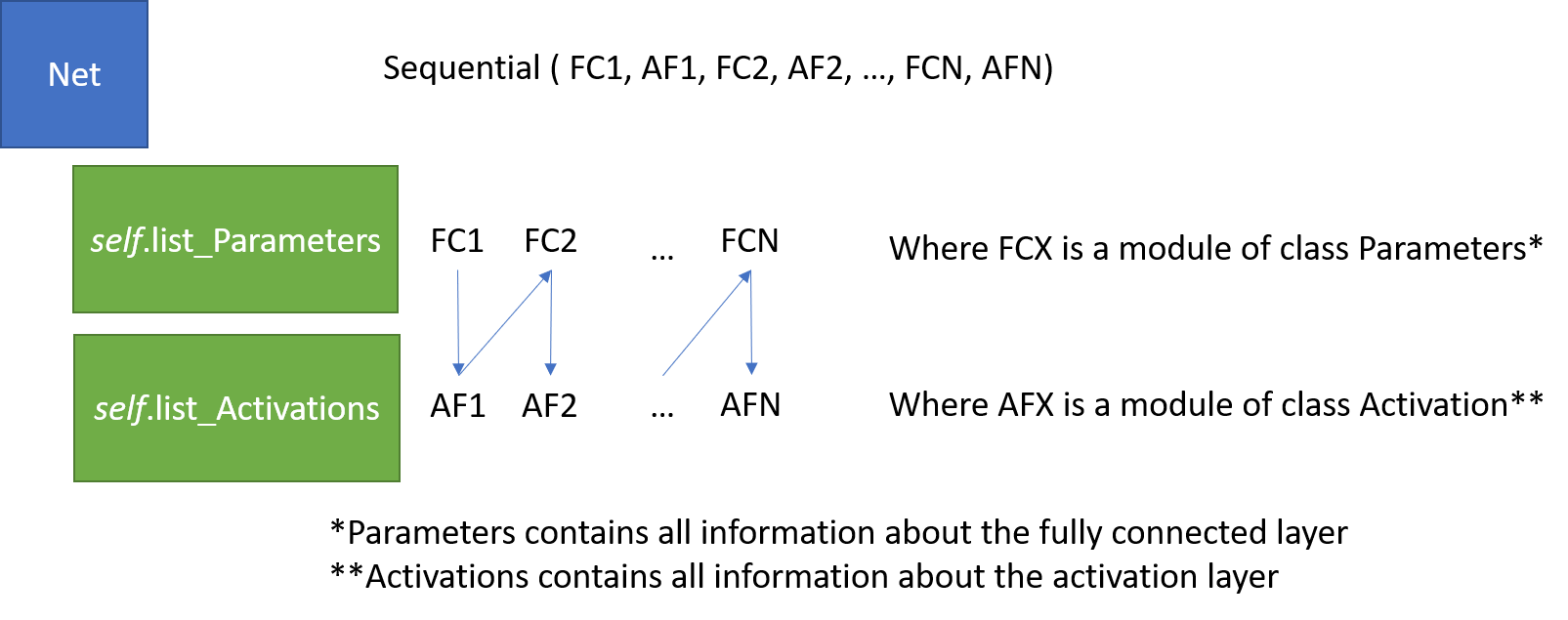


Figure 3:Vue d'ensemble structure

Chaque module Parameter et Activation, stock en mémoire les données nécessaires à la backpropagation lors du forward. Les seules valeurs à retenir sont dans la classe Activation :

* ds\_da : dérivée de la sortie de la fonction d’activation évalué en s.
* da\_dw : la dérivée de la fonction linéaire par rapport à ces poids (directement l’entrée de la fonction d’activation : x (couche précédente))..

Quant au backward de la classe Net, il prend en entrée, la dérivée de la fonction perte évalué à la sortie du forward de Net (module Loss). À l’instar de forward, il appelle successivement dans l’ordre inverse (droite à gauche) les fonctions backward des éléments des champs Paramètre et Activations.

En utilisant les valeurs stockées (dans Activation), les gradients de la fonction perte en fonction des coefficients des combinaisons linéaire de la couche linéaire, situé avant la couche d’activation, sont retourné par la fonction backward des éléments du champs Activation de Net.

De cette manière on peut accumuler les gradients dans la class Net, pour finalement mettre à jour les poids du réseau à chaque nouveau mini-batch dans la boucle d’entrainement du fichier ‘test.py’.

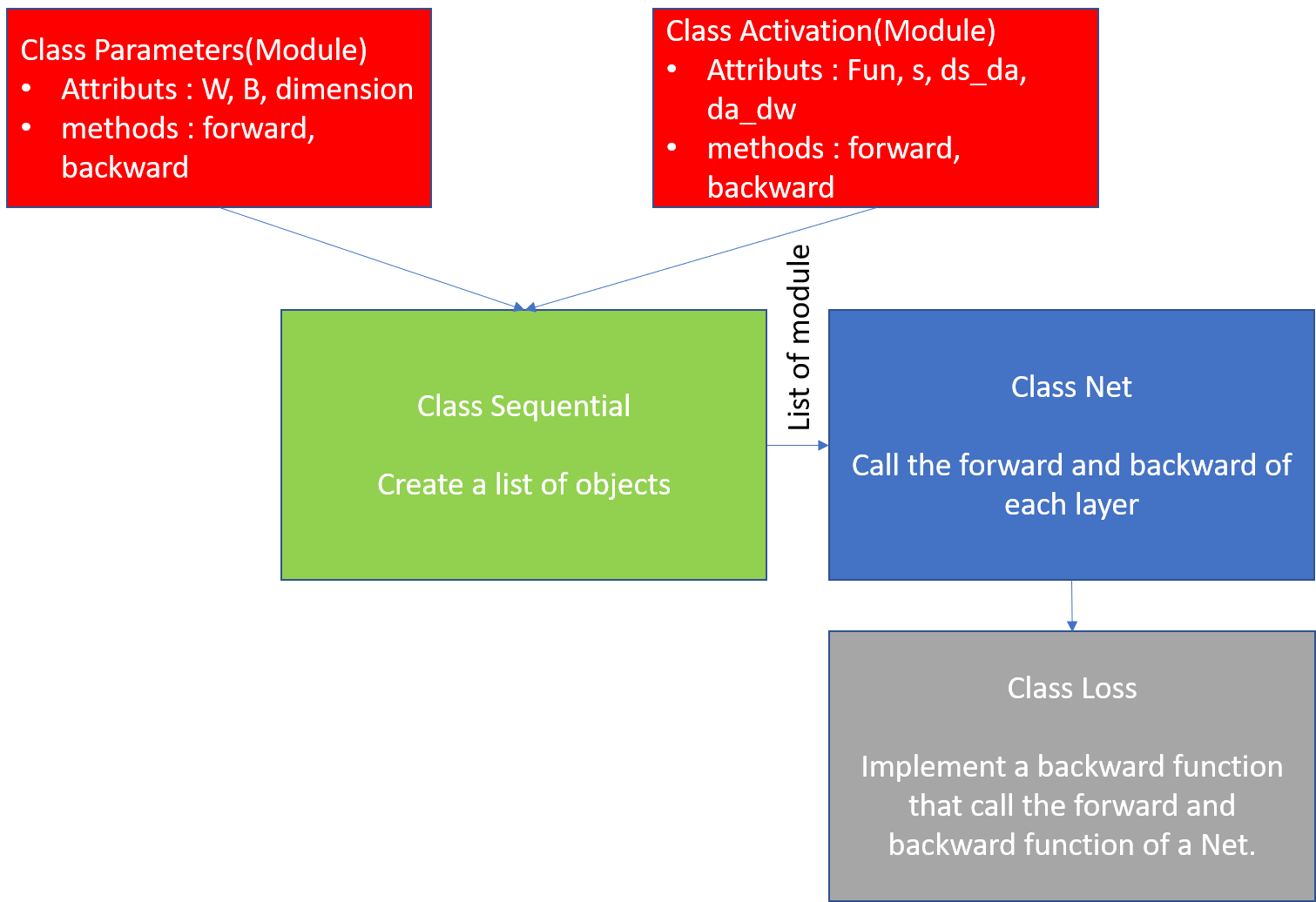


Figure 4: Global structure