Project 2: Noise2Noise writing your own blocks

Estelle CHABANEL 284197, Clara LE DRAOULEC 287280, Hugo WITZ 284336
EE-559 - Deep Learning, EPFL, Switzerland

I. INTRODUCTION

In this project we built a denoiser model but with our own modules implemented from scratch using only pytorch's tensor operations and the standard python library. A more detailed implementation procedure of the modules will be further given in this report.

II. IMPLEMENTATION OF MODULES

Following the structure of the modules given in the project description, we used the following framework containing three functions for the Class Module :

- **Forward(self, input)** : this function takes as input a tensor and return a tensor.
- **Backward(self, gradwrtoutput)** : this function takes as input a tensor containing the gradient of the loss with respect to the module's output and returns a tuple of tensor that contains the gradient with respect to the loss of the module's input and parameters.
- **Param(self)** : this function returns a list of pairs of tensor and gradient of each parameter, if any in the module.

A. Sequential

In order to assemble our different modules to create the network, we have implemented the sequential module. By taking a list of those modules, the sequential allows to combine them. In the forward pass, it calls the forward function of the called module with the given inputs. The resulting output is used to feed the next called module and so on until the end of the list. The backward pass has the same functioning but it calls the backward of the called modules in the reverse order and with the derivative of the loss function as input.

B. Activation functions

As requested, we have built two activation functions : ReLU and Sigmoid.

1) ReLU: For the ReLU function, we clamp all the negatives values to zero in the forward path and in the backward path we compute the derivative of the input and multiply it by the gradient with respect to the output. The ReLU module has no parameter.

2) Sigmoid: We compute the sigmoid function in the forward path using its definition and the derivative in the backward path. As ReLU, the sigmoid module is parameterless.

C. Convolution layer

First, we have initiated our weights and bias that follow a normal distribution, and then we initialized the derivatives as zero tensors. With the hints in the project description, we have implemented the forward path of the convolution by writing the convolution as a linear function, applying our logic for linear operations, and reinterpreting the output as a convolution.

However, we have used for loops for the backward propagation. Indeed, even though it has a higher computational cost, it appeared simpler and clearer to us. The backward function takes as input the gradient with respect to the output and returns the gradient with respect to the input and the parameters. The parameters return two pairs of tensors : the weights and the bias and their respective gradients.

D. Transposed convolution layer

We initialize the module in the same way as the convolution. Then, the forward function compute the transposed convolution of the input using the weights and bias and returns the output. The backward function computes the gradient with respect to the input and the parameters, in the idea of the one in the convolution module.

E. Upsampling layer

The upsampling layer could have been implemented using the transposed convolution module. However, we chose to implement it from scratch because this approach was the easiest for us to understand. In the forward function, the module returns an upsampled tensor from the input, where rows and columns have been added according to the *size* argument. Then, the backward function takes as input the gradient with respect to the output and return the gradient with respect to the input, which contains the same value but whose dimension is lower. In the way we built this module, it contains no parameters.

F. Loss function : MSE

The Mean Squared Error module computes the loss of the output w.r.t the target values in the forward function and computes the derivative, that is then fed to the network's backward pass in the backward function.

G. Optimizer : SGD

The SGD module takes as input a list of model's outputs and a learning rate. Instead of using the forward/backward structure as for the other modules, we decided for simplicity to keep the classic structure of the SGD. Then, we have a step function which updates the parameters w.r.t the gradient while the zero grad function resets the gradient of the parameters to zero.

III. Results

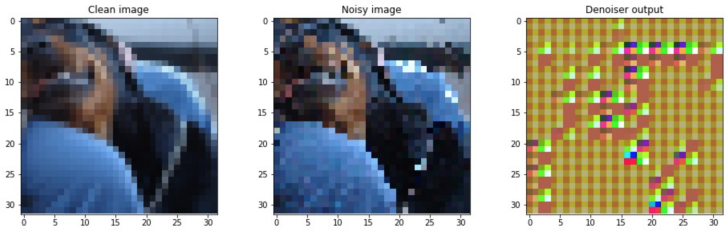

Figure 1: Denoiser output

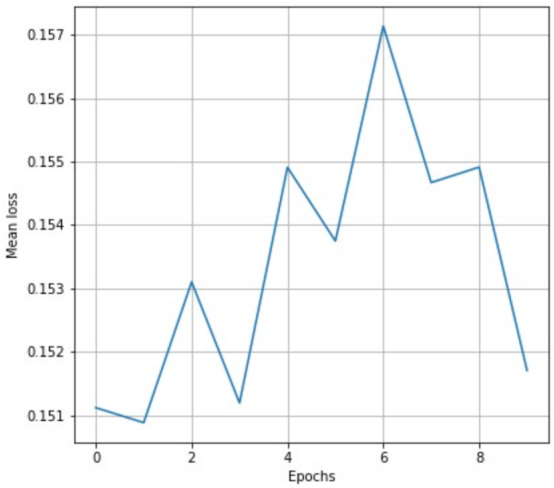

Figure 2: Evolution of the loss during training

As we can observe, the denoiser gives a very bad prediction, even after multiple testing with various parameters. This is confirmed by the loss evolution which is chaotic compared to the one shown in the project 1.

IV. Discussion

Our model performed poorly on the denoising task. The loss is indeed not decreasing as expected and this may be due to several factors :

- Errors in the implementation in one of our layers or in the optimizer,
- Non optimal parameters,
- Errors in the train model that may result in wrong inputs passed through the functions.
- The SGD optimizer might not have the higher performance for this task.

As we expected, the computational cost of our modules is quite high due to the use of for loops.

However, even though our deep learning framework did not give satisfying results, it has allowed us to have a better comprehension of the Pytorch modules. We have a clearer understanding of what is happening during the training of a neural network and this project has helped us apprehending the theory of the course.