École Polytechnique Fédérale de Lausanne

Deep Learning (EE-559)

Mini-project 2

Group Members:

Hugo Miranda Queiros

SCIPER: 336706

Francesco Nonis

SCIPER: 300540

Erfan Etesami

SCIPER: 337448

Spring 2022

1 Introduction

In the first mini-project of the *Deep Learning (EE-559)* course, we built a deep network denoiser using the PyTorch framework. The goal of the second mini-project was to build our own PyTorch inspired framework and to use it to build a simple predetermined denoiser to validate our framework. More specifically, we were tasked with implementing the following features: convolutions, transposed convolutions, ReLU and Sigmoid activation functions, an SGD optimizer, an MSE loss function, and a container (similar to the *sequential* container of PyTorch) to allow us to easily stack the previous modules to build a network.

2 Module's Structure

Each of the modules, except for the SGD module, is implemented in a class inheriting from the *Module* class which contains the three basic functions of *forward*, *backward*, and *param*. The *forward* and *backward* functions compute the forward and backward pass of a module respectively. The *param* function returns the parameters as well as their respective gradient.

2.1 Backward

The backward function receives the gradient of the loss with respect to the module's output and computes the gradient of the loss with respect to the module's input. It must also accumulate the gradient with respect to the parameters of the module. To do so, it relies on the derivation chain rule which can be written for our case as:

$$\frac{\partial Loss}{\partial input} = \frac{\partial Loss}{\partial output}\frac{\partial output}{\partial input} \tag{1}$$

This means that, for each module, we must only compute the derivative of the output with respect to the input and multiply the result by the derivative of the loss with respect to the output which is given as an argument to the backward function.

3 Modules implementation

3.1 Initialization of the Parameters

A key element to make any deep network reach a good optimum is the initialization of the weights and bias. Among the various initialization techniques that exists, we chose to implement the same one as PyTorch (described in their documentation of the convolution and transposed convolution modules [1]).

3.2 MSE, ReLU and Sigmoid Module

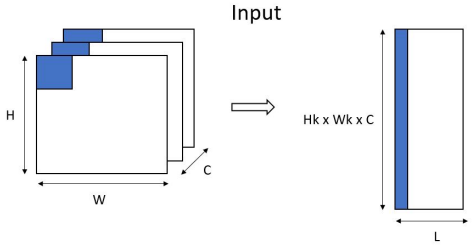
To avoid unnecessary repetition, the *MSE* loss, *ReLU*, and *Sigmoid* modules are treated together since the same logic was used to implement them. Their implementation is quite straightforward because all of them consist of a simple transcribing of the analytical expression of their respective formula for the forward pass and of the derivative of the module's output with respect to the input needed for the backward pass as stated in equation 1. Those expression are summarized in the table 1.

3.3 Convolution Module

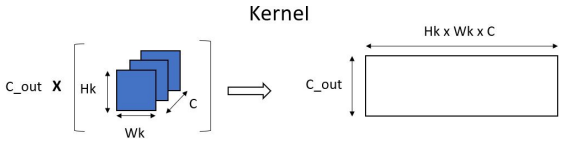
To compute the forward pass of the convolution module, the convolution is reformulated as a matrix product. In order to do so, the kernel and the input must be reshaped, as shown in fig. 1. This is done by using the *unfold()* and the *view()* functions. After reshaping the data, we can simply do a matrix product between the reshaped kernel and the reshaped input. The output must then be reshaped as an $N \times C_{out} \times H_{out} \times W_{out}$ 4D tensor.

module	forward pass	$\frac{\partial \sigma}{\partial x}$
MSE error	$\frac{1}{M}\sum_{i=1}^{M}(x_i - x_{target_i})^2$	$\frac{2}{M}(x - x_{target})$
ReLU	$\sigma(x) = max(0, x)$	$\begin{cases} 0 \text{ if } x < 0 \\ 1 \text{ otherwise} \end{cases}$
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	$Sigmoid(x)(1 - Sigmoid(x))$

Table 1: Forward pass with their respective derivative.



(a) Reshaping of input with unfold().



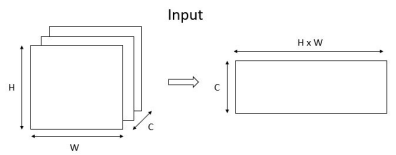
(b) Reshaping of kernel with view(): C_{out} is the number of output channel of the convolution.

Figure 1: Reshaping of the kernel and of the input (example for N = 1, minibatch of size 1).

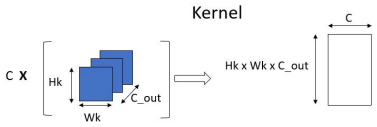
Concerning the backward pass, the result of the input is straightforward since $\frac{\partial Loss}{\partial input}$ is the transposed convolution of $\frac{\partial Loss}{\partial output}$. For the weights, the trick was just to associate them to all their corresponding inputs and outputs' gradients by playing with *unfold()* and *view()* functions. For the biases, the result is again straightforward because $\frac{\partial Loss}{\partial bias}$ is equal to the sum of $\frac{\partial Loss}{\partial output}$ in its corresponding output channel.

3.4 Transposed Convolution Module

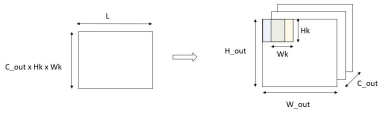
As for the convolution module, the transposed convolution operation was reformulated as a matrix product by reshaping the tensors as depicted in fig. 2. To compute the transposed convolution, we first compute the matrix product of the reshaped kernel with the reshaped input thanks to *view()*. This results in a $N \times (C_{out} \times H_k \times W_k) \times L$ 3D tensor. Then, the *fold()* function is used to reshape this tensor into a $N \times C_{out} \times H_{out} \times W_{out}$ 4D tensor.



(a) Reshaping of the input with view().



(b) Reshaping of kernel with view(): C_{out} is the number of output channels.



(c) Folding: The overlapping elements of each block are summed.

Figure 2: Reshaping of the kernel and of the input (example for N = 1, minibatch of size 1).

Concerning the backward pass, the result of the input is straightforward since $\frac{\partial Loss}{\partial input}$ is the convolution of $\frac{\partial Loss}{\partial output}$. For the weights, the process is similar as what was explained before for the convolution. For the biases, the computation is exactly the same as for the convolution.

3.5 Sequential Module

The sequential module is a container that would allow us to easily stack layers to build a network exactly as the *nn.Sequential* module from PyTorch. In order to be able to receive a variable number of

layers to build the network, the following syntax *__init__(self, *layers)* is used. The forward pass of the whole network is computed by computing the forward pass of each layer in the right order (i.e. the output of a layer is the input of the next layer). The backward pass is done by calling the *backward* function of each layer by going through them in reversed order (i.e. the output of the *backward* function of a layer is the input of the previous layer) which brings about back-propagating the gradient from the last to the first layers.

3.6 SGD Module

The SGD module is the only module which does not follow the structure of the *Module* class presented in section 2. Indeed, since it is our optimizer, it does not need a forward or backward function. It only needs a step function which updates the parameters according to learning rate and the gradient of the loss with respect to the parameters. The update rule of a SGD optimizer is the following:

$$w_{t+1} = w_t - \eta \nabla l(w_t) \tag{2}$$

$$b_{t+1} = b_t - \eta \nabla l(b_t) \tag{3}$$

The parameters which must be optimized and the learning rate are given as an argument in the initialization of the SGD. By initializing a variable with this line of code *self.parameters = parameters*, it will only create another *nickname* for the same variable, meaning that by modifying the variable *self.parameters* in place, we will also modify *parameters*. Consequently, we can update all the parameters of the network directly within this function.

4 Results

4.1 Comparison with PyTorch

All the implemented modules have been independently compared to the results of PyTorch to validate our implementation using the *torch.testing.assert_allclose()* function. Note that, when comparing large values ($> 10^3$), a warning is raised by the *torch.testing.assert_allclose()* since the value from our implementation differs from the one of PyTorch in the order of 10^{-5}. Given that, this is really small compared to the values that are compared and that this does not happen systematically. It is probably due to rounding errors and is hence negligible.

The major difference between our framework and PyTorch was the execution time. Our implementation took 16.8 s to execute one epoch while this took 2.7 s when using PyTorch and running it on the GPU. We think this difference in performance is due to PyTorch's using of *autograd* which is more effective than our simple back-propagation implementation.

4.2 Performance of the test denoiser

By implementing the basic denoiser that was shown in the instructions of the mini-project and using our framework, we reached a PSNR of 15.97 dB (with the PSNR of `test.py`) after 10 minutes of training on GPU. Very similar results were obtained when implementing the same denoiser in PyTorch. Since this result was not very satisfying, we tried to see if this could be improved. With a simple RED-Net [2] of 8 layers, we managed to reach 23.6 dB of noise reduction in less than 10 minutes of training on GPU. If it is wished to run this architecture instead of the one proposed in the instruction, the *denoiser_bonus.py* file in the *others* folder of *Miniproject2* should be renamed into *denoiser.py*, and, conversely, the *denoiser.py* should be renamed into something different.

References

[1] PyTorch convolution documentation. `https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html?highlight=conv2d#torch.nn.Conv2d`. [Online; accessed 25-May-2022].

[2] Mao, X., Shen, C., & Yang, Y. B. (2016). Image restoration using very deep convolutional encoder-decoder networks with symmetric skip connections. *Advances in neural information processing systems, 29.*