Project 2: Mini Framework for Noise2Noise

Manon Béchaz, Thomas Castiglione, Emilien Seiler
EE-559 Project 2, EPFL

***Abstract*—Current Deep Learning frameworks offer great libraries with a wild amount of application areas. This report proposes a concrete construction, from scratch, of an alternative mini framework for denoising images via Noise2Noise. The constructed framework has allowed to construct a functional network, reaching after 30 minutes of training a PSNR of 22.50 dB despite a simple architecture.**

I. Introduction

In addition to allowing the development of an efficient model for image denoising, the first project highlighted the ease to develop such complex models with `PyTorch` [1]. To further understand the convenience that libraries like `PyTorch` provide, this second project aims at designing from scratch a mini deep learning framework using only the basic `PyTorch`s tensor operations. The latter should provide the necessary tools to build convolutional networks and to optimize them using MSE and SGD.

II. Modules

In this section, we introduce the structures of the modules in our framework as well as the mathematical principles of their forward and backward propagation processes. As for classical frameworks, each module is equipped with a `forward` and a `backward` method. The `forward` method takes an input x and returns $f(x)$, with f the operation represented by the module. The `backward` method takes as input the gradient of the previous layer, and computes the gradient with respect to the current layer. When they exist, this method also calculates the gradient with respect to the parameters of the layer. In what follows, X denotes the input of the layer, O the output and L the loss.

A. Convolutional layer

Mathematically, a convolution is a linear application from $\mathbb{R}^{C_{in}\times H\times W}$ to $\mathbb{R}^{C_{out}\times H'\times W'}$, with C_{in} and C_{out} the input and output channel dimensions, $(W \times H)$ the spatial dimensions of the image, and $(W' \times H')$ the dimensions of the convolved image. H' and W' are computed as follows [2]:

$$W' = \left\lfloor \frac{W + 2 * \text{padding}_1 - \text{dilation}_1(k_1 - 1) - 1}{\text{stride}_1} + 1 \right\rfloor \quad (1)$$

$$H' = \left\lfloor \frac{H + 2 * \text{padding}_2 - \text{dilation}_2(k_2 - 1) - 1}{\text{stride}_2} + 1 \right\rfloor, \quad (2)$$

with k_1 and k_2 the spatial dimensions of the kernel. The convolution can thus easily be implemented, after some reshaping, as a linear layer. The `forward` pass is schematically represented in Figure 1.

The implementation of the `backward` pass also relies on the interpretation of the convolution as a linear application. Indeed,

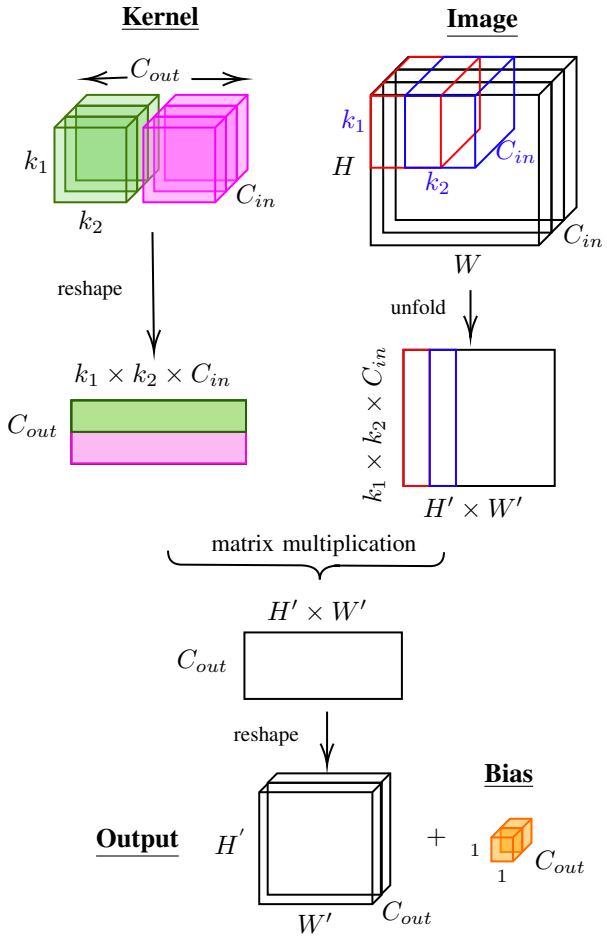

Fig. 1. Implementation of the convolution as a linear application.

as described in Figure 1, the convolution is written as a matrix multiplication:

$$O_{\text{resh.}} = K_{\text{resh.}} \times X_{\text{resh.}} \ (+\text{Bias}), \quad (3)$$

with $O_{\text{resh.}}$, $K_{\text{resh.}}$ and $X_{\text{unf.}}$ the reshaped output and kernel and the unfolded input image. Using the chain rule and the previously introduced notation, one has:

$$\frac{dL}{dX_{\text{unf.}}} = \frac{dL}{dO_{\text{resh.}}}\frac{dO_{\text{resh.}}}{dX_{\text{unf.}}} = K_{\text{resh.}}^T \frac{dL}{dO_{\text{resh.}}} \quad (4)$$

$$\frac{dL}{dK_{\text{resh.}}} = \frac{dL}{dO_{\text{resh.}}}\frac{dO_{\text{resh.}}}{dK_{\text{resh.}}} = \frac{dL}{dO_{\text{resh.}}} X_{\text{unf.}}^T. \quad (5)$$

This way, the gradient with respect to the input and the gradient with respect to the kernel are computed from the reshaped and unfolded quantities, and are eventually once again reshaped to match the initial format of the input, resp. kernel. Turning to the bias, the latter is added after the final reshaping. The computation of the gradient with respect to this bias is thus identical to the linear case and goes back to summing the gradient with respect to the previous layer along the batch size and the spatial dimensions.

B. Nearest Neighbor Upsampling

The Nearest Neighbor Upsampling layer consist in interpolating the input image by selecting the value of the nearest point. From a mathematical point of view, the Nearest Neighbor Upsampling can be written as:

$$O_{\text{scale}_1 i+k_i, \text{scale}_2 i+k_j} = x_{i,j} \qquad \text{for } k_{i,j} = 0, ..., \text{scale}_{i,j} - 1, \tag{6}$$

with $\text{scale}_{1,2}$ the scaling factors. Note that the previous expression does not consider the channel, the latter can however be generalized for multiple channels by simply performing the same operation separately on each channel.

Regarding the backpropagation, the chain rule implies that

$$\frac{dL}{dx_{i,j}} = \sum_{m,n} \frac{dL}{dO_{m,n}} \frac{\partial O_{m,n}}{\partial x_{i,j}} \tag{7}$$

$$= \frac{dL}{dO_{m,n}} \delta_{i,\lfloor m/\text{stride}_1 \rfloor} \delta_{j,\lfloor n/\text{stride}_2 \rfloor}, \tag{8}$$

where we used equation (6) to express $\frac{\partial O_{m,n}}{\partial x_{i,j}}$. This expression is equivalent to summing the upstream derivative on pads of size $\text{scale}_1 \cdot \text{scale}_2$. The forward and backward processes are sketched in Figure 2.

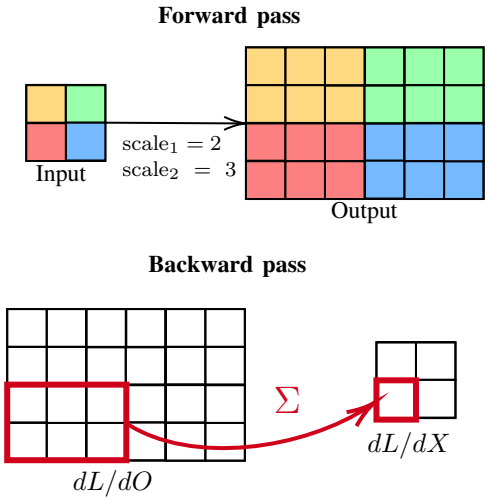

Fig. 2. Forward and Backward pass of the Nearest Neighbor Upsampling.

C. Upsampling layer

Usually implemented via the transposed convolution, the Upsampling layer can also be constructed using a nearest neighbor upsampling followed by a convolution. More precisely, a transposed convoltion such as

```
tconv = TransposeConv(Cin,Cout,kernelsize = 3,strid
    =2,padding=1)
y = tconv(x)
```

can be replaced by

```
ups = NearestUpsampling(scalefactor=2)
conv = Conv2d(Cin,Cout,kernelsize=3,padding=1)
u = ups(x)
y = conv(u)
```

It is this implementation of convolution that has been used in this project.

D. Activation Functions

Activation functions allow the addition of non linearity in the network. Let M be a Module representing an activation function f. Using the previously introduced notation:

- the `forward` method of M returns $f(X)$,
- the `backward` method returns

$$\frac{dL}{dX} = \frac{dL}{dO}\frac{dO}{dX} = \frac{dL}{dO} f'(X). \tag{9}$$

Different activation functions have been implemented:

- `ReLU`: $f(x) = \max(0, x)$, $f'(x) = \begin{cases} 1, & x > 0 \\ 0, & \text{otherwise} \end{cases}$,
- `Sigmoid`: $f(x) = (1 + e^{-x})^{-1}$, $f'(x) = f(x)(1 - f(x))$,
- `Tanh`: $f(x) = \tanh(x)$, $f'(x) = \cosh(x)^{-2}$.

E. Sequential Container

The purpose of the `Sequential` module is to combine several modules in a basic sequential structure. The latter consist of a list of successive modules. Its `forward` and `backward` methods simply apply successively the `forward` and `backward` methods of its constituent modules.

III. Loss Function

The Mean Squared Error (MSE) supplied in this framework is perhaps the simplest and most common loss function. The MSE is given by

$$L = \frac{1}{n} \sum_{i=1}^{n} (O - T)^2, \tag{10}$$

with O the output of the network, T the target, and n the batch size. Turning to the backward pass, the derivative of the MSE, first step of the back propagation, is given by

$$\frac{dL}{dO} = \frac{2}{n} \sum_{i=1}^{n} (O - T). \tag{11}$$

IV. Optimizer

Stochastic Gradient Descent (SGD) is implemented in this framework.

V. Framework testing

The Framework is tested on the following network:

```
model = Sequential(Conv2d(input_channels, nbchannels1
    , kernel_size=kernel_size,stride=2, padding=1),
    ReLU(),
    Conv2d(nbchannels1, nbchannels2, kernel_size=
    kernel_size, stride=2,padding=padding),
    ReLU(),
    NearestUpsampling(2),
    Conv2d(nbchannels2, nbchannels1, kernel_size=
    kernel_size, padding=padding),
    ReLU(),
    NearestUpsampling(2),
    Conv2d( nbchannels1, output_channels, kernel_size
    =kernel_size, padding=padding),
    Sigmoid())
```

The general structure is fixed, the only degrees of freedom are the number of channels of the convolutions as well as their kernel size. The padding is adapted to ensure that the output has the same dimension as the input. The methodology followed is identical to the one presented in the first project [1].

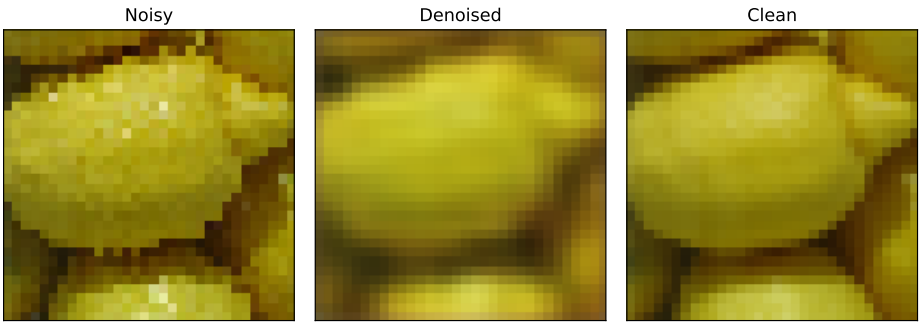

Fig. 3. Typical example of the denoising of an image with the final network, and comparison with the corresponding clean image. For this image, the model achieved a PSNR of 25.52 dB.

A. Optimal number of channels

The number of channels of the convolution layer can have a non-negligible impact on the performance of the network. Finding appropriate values is therefore essential.

For a kernel of size 3, the network is trained on 1000 images and 10 epochs for numbers of channels `nbchannels1`, `nbchannels2` ranging in $\{32, 64, 128, 256\}$. The resulting networks are then tested on 100 images of the validation set. The obtained averaged PSNR are provided in Figure 4, with respect to the total number of parameters of the network.

In view of these results, the retained values are 256 for the first channel number, and 32 for the second. Indeed, the latter achieve a high PSNR with a limited number of parameters, thus allowing to reach, for an equal training time, better performances than a network with more parameters and therefore longer to train.

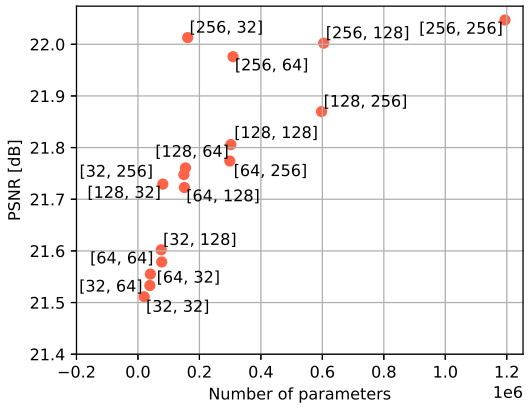

Fig. 4. PNSR with respect to the number of parameters for models with different number of channels for the convolution. The bracketed numbers correspond to the first and second number of channels. The models are trained on 1000 images and 10 epochs.

B. Optimal kernel size

The second degree of freedom of the proposed network is the size of the convolution kernel. For the numbers of channels previously determined, the network is tested for different kernel sizes. The results are summarized in Table I.

It is undeniable that a kernel of size 3 is more efficient than kernels of larger size. The latter reach a higher PSNR, with a smaller number of parameters and hence reduced training time.

Kernel size	Averaged PSNR on the validation set [dB]	Number of parameters
3	22.36	161827
5	21.50	448547
7	21.90	878627

TABLE I
Averaged PSNR on the validation set and number of parameters of the model for different kernel sizes.

C. Final network

With the optimal kernel size (3) and number of channels (256 and 32) previously established, the network is trained 30 minutes, i.e. on 3000 images and 10 epochs. The idea behind this short training is to enable the comparison with the models developed in the first project [1], whose training times were limited to 10 minutes on a GPU and 30 min on a CPU.

The model achieved a mean PSNR of 22.50 dB on the validation set. An example of denoised image obtained with the network is given in Figure 3.

As expected, the results are below the ones obtained in the first project. However, the structure of the network is also much simpler. Working with similar architectures could have led to better results, that are approaching the results of the first project, but would have required much larger training times. Indeed, although the framework presented here is functional, it remains much less optimized and performant than that of standard libraries such as `pytorch`.

VI. Conclusion

To conclude, the framework developed has allowed the construction of a working denoising network. Despite the simple architecture imposed, the network surprisingly produces pretty good results, reaching correct values of PSNR after only 30 minutes of training. Nonetheless, the produced network is not as well-optimized as traditional Deep Learning frameworks, thus yielding in larger training time.

References

[1] Emilien Seiler Manon Béchaz, Thomas Castiglione. Noise2noise using the standard pytorch framework.
[2] Conv2d - pytorch 1.11.0 documentation, https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html.