Mini-Project 2

Anya-Aurore Mauron, Louis Poulain--Auzéau and Servane Lunven
Deep Learning, EPFL Lausanne, Switzerland

I. Introduction

In project 1, the purpose was to denoise images with the help of the powerful framework offered by the library Pytorch [3] and based on the Noise2Noise paper [2]. The efficiency of deep convolutionnal networks, more particularly autoencoders, has been proven in the past project. In this second project, the aim does not change but one has to implement the framework from scratch, with the constraint that only the basic functions of Pytorch and the standard Python libraries can be used.

II. Implementation

Autoencoders are a specific type of feedforward neural networks where the input has the same shape as the output. It is composed of two main parts: the encoder and the decoder. In the encoder, the input data is transformed into a representation of lower dimension. It is thus trained to capture the most important part of the input images. The decoder then reconstructs the inputs images using only the encoder results. The architecture and the classes used in this project are described in Table I.

Layer	Parameters
`Conv2d` + `ReLU`	$C_{in} = 3$, $C_{out} = C$ $k_{size} = 3$ $stride = 2$, $padding = 1$
`Conv2d` + `ReLU`	$C_{in} = C$, $C_{out} = C$ $k_{size} = 3$ $stride = 2$, $padding = 1$
`NearestUpsampling`	$scale_f = 2$
`Conv2d` + `ReLU`	$C_{in} = C$,$C_{out} = C$ $k_{size} = 3$ $stride = 1$, $padding = 1$
`NearestUpsampling`	$scale_f = 2$
`Conv2d` + `Sigmoid`	$C_{in} = C$, $C_{out} = 3$ $k_{size} = 3$ $stride = 1, padding = 1$

TABLE I: Architecture of the model used in the project.

The parameter C will be fine tuned in the following part. These classes, as well as the class `Sequential` which contains the set of modules forming the model, inherit from the super class `Module`. The latter is composed of a forward and backward functions, which are redefined in each sub-class. In the forward function of the class `Sequential`, intermediate variables are sequentially calculated and stored from the input layer to the last one.

In the backward function, the backpropagation algorithm is implemented. It consists in sequentially calculating and storing the gradients of intermediate variables and parameters from the calculation of the loss L, implemented in the class `MSE`, to the first layer. For this, the chain rule is used. Once these gradients are calculated, the parameters are updated using a stochastic gradient descent update rule, implemented in the `SGD` class. For example the weights of i^{th} convolutional layer $w^{(i+1)}$ will be updated as:

$$w^{(i+1)} = w^{(i+1)} - l_r * \frac{\partial L}{\partial w^{(i+1)}}$$

where l_r is the learning rate, set to 0.1 in the project. Then the chain rule gives

$$\frac{\partial L}{\partial w^{(i+1)}} = \frac{\partial L}{\partial y} \cdots \frac{\partial x^{(i+1)}}{\partial x^{(i)}} \frac{\partial x^{(i)}}{\partial w^{(i+1)}}. \tag{1}$$

A. `Conv2d`

Let $C_{in} \times H \times W$ and $C_{out} \times H_{out} \times W_{out}$ be the number of channels, the height, the width of the input I, respectively the output. The **forward pass** of Convolutional 2D layers consists in applying a convolution on a input image I with a weight vector w, called *kernel*. The vector w of size $C_{out} \times C_{in} \times k_0 \times k_1$, where $k_0 \times k_1$ is the size of the kernel. Potentially, the convolution is summed by a value b of size C_{out}, called *bias*. Let $s \in N^*$ be the *stride*, such that we sample every s pixel in each direction [1]. The output Z, of size $C_{out} \times H_{out} \times W_{out}$, of this operation is given by:

$$\begin{aligned} Z(k,i,j) &= (I * w)(k,i,j) \\ &= b_k + \sum_{c=1}^{C_{in}} \sum_{x=1}^{H} \sum_{y=1}^{W} I(c, i \cdot s + x, j \cdot s + y) w(k,c,x,y) \end{aligned}$$

where $k = 1, ..., C_{out}$, $i = 1, ..., H_{out}$ and $j = 1, ..., W_{out}$.

We also add a *padding* parameter to the implementation. It consists in adding extra pixels around the boundary of the input image. The shape of the result is thus:

$$H_{out} = \lfloor \frac{h + 2 * padding[0] - (ksize[0] - 1) - 1}{stride[0]} + 1 \rfloor$$

$$W_{out} = \lfloor \frac{w + 2 * padding[1] - (ksize[1] - 1) - 1}{stride[1]} + 1 \rfloor$$

To avoid loops in `Conv2d`, which would have considerably increased the running time, a matrix multiplication method has been used to code it. It is described in Figure 1.

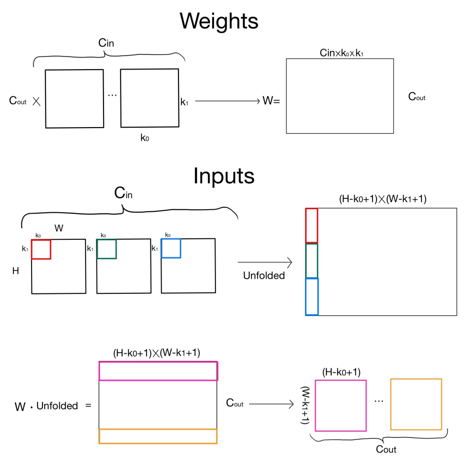

Fig. 1: Coding steps of the forward pass of `Conv2d`.

The **Backward pass** results from [1]. The derivative of the loss $L(I, w)$ with respect to $w(k, c, x, y)$ (with $k = 1, ..., C_{out}$, $c = 1, ..., C_{in}$, $x = 1, ..., k_0$, $y = 1, ..., k_1$) is given by:

$$\begin{aligned} &\frac{\partial}{\partial w(k,c,x,y)} L(I,w) \\ &= \sum_{\hat{c}=1}^{C_{out}} \sum_{i=1}^{H_{out}} \sum_{j=1}^{W_{out}} \frac{\partial L(I,w)}{\partial Z(\hat{c},i,j)} \frac{\partial Z(\hat{c},i,j)}{\partial w(k,c,x,y)} \\ &= \sum_{i=1}^{H_{out}} \sum_{j=1}^{W_{out}} \frac{\partial L(I,w)}{\partial Z(c,i,j)} I(k, i \cdot s + x, j \cdot s + y) \end{aligned}$$

since $\frac{\partial}{\partial w(k,c,x,y)} Z(\hat{c},i,j) = I(l, i\cdot s + x, j \cdot s + y)$, if $\hat{c} = c$, and is zero otherwise. In the same manner, the derivative of the loss with respect to the bias is (with $k = 1, ..., C_{out}$):

$$\frac{\partial}{\partial b_k} L(I,w) = \sum_{x=1}^{H_{out}} \sum_{y=1}^{W_{out}} \frac{\partial L(I,w)}{\partial Z(k,x,y)}$$

since $\frac{\partial}{\partial b_k} Z(k,x,y) = 1$. By a change of variables, we can write $Z(k,i,j) = b_k + \sum_c \sum_x \sum_y I(c,x,y) w(k,c,x-i\cdot s, y - j \cdot s)$, which implies

$$\frac{\partial Z(k,i,j)}{\partial I(c,x,y)} = w(k,c,x-i\cdot s, y-j\cdot s)$$

with $c = 1, ..., C_{in}$, $x = 1, ..., H$ and $y = 1, ..., W$. Thus the derivative of the loss with respect to the input image is

$$\begin{aligned}\frac{\partial L(I,w)}{\partial I(c,x,y)} &= \sum_{k=1}^{C_{out}} \sum_{i=1}^{H_{out}} \sum_{j=1}^{W_{out}} \frac{\partial L(I,w)}{\partial Z(k,i,j)} \frac{\partial Z(k,i,j)}{\partial I(c,x,y)} \\ &= \sum_{k,i,j} \frac{\partial L(I,w)}{\partial Z(k,i,j)} w(k,c,x-i\cdot s, y-j\cdot s).\end{aligned}$$

For the implementation, we again avoid loops and use the `fold` and `unfold` pytorch functions. To compute the sums over the height and the width, we use matrix products, hence the matrix product along the side of size $H_{out} \times W_{out}$, see Figure 2.

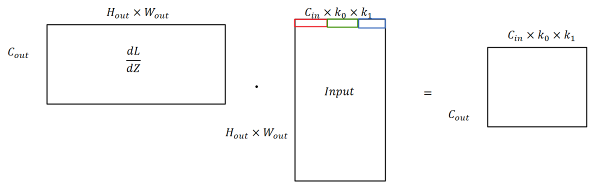

Fig. 2: Backward pass for the convolution: computation of $\frac{\partial}{\partial w} L(I,w)$ by matrix product.

To implement the derivative of the input with respect to the input, refer Figure 3. First a matrix product is done for summing over the channels. After transposing the product, the function `unfold` allows to compute the sums over the height and the width, and retrieve the right format.

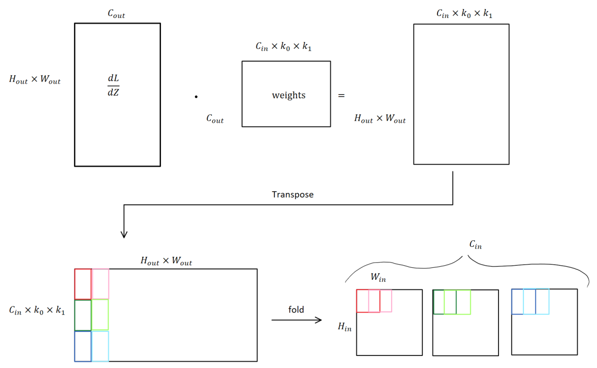

Fig. 3: Backward pass for convolution: computation of $\frac{\partial}{\partial I} L(I,w)$ by matrix product.

B. `Nearest Upsampling`

The **forward pass** consists in enlarging the input image by a scale factor s by assigning to the new pixels the value of the corresponding nearest pixel of the input I. The output Z of size $C_{in} \times H_{out} \times C_{out}$ is given by:

$$Z(c,i,j) = I(c, i \,\%\, s, j \,\%\, s)$$

where the sign % is the modulo function.

The **backward pass** consists of computing the derivative of L with respect to the input I. It is given by

$$\begin{aligned}\frac{\partial L(I)}{\partial I(c,x,y)} &= \sum_{\hat{c}=1}^{C_{in}} \sum_{i=1}^{H_{out}} \sum_{j=1}^{W_{out}} \frac{\partial L(I)}{\partial Z(\hat{c},i,j)} \frac{\partial Z(\hat{c},i,j)}{\partial I(c,x,y)} \\ &= \sum_{\hat{c},i,j} \frac{\partial L(I)}{\partial Z(\hat{c},i,j)} \delta_{c,\hat{c}} \delta_{x, m \,\%\, s} \delta_{y, n \,\%\, s}\end{aligned}$$

where δ is the Kronecker delta.

C. Activation layers

The **forward pass** of the activation functions is given by:

$$\texttt{ReLU:}\ Z_r(c,i,j) = \begin{cases} I(c,i,j) & \text{if } I(c,i,j) > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\texttt{Sigmoid:}\ Z_s(c,i,j) = \frac{1}{1 + e^{I(c,i,j)}}$$

where $Z_r \coloneqq \texttt{ReLU}(I)$, $Z_s \coloneqq \texttt{Sigmoid}(I)$.

In the **backward pass**, the derivative of the output Z_r and Z_s with respect to the input I are given by

$$\frac{\partial Z_r(c,i,j)}{\partial I(\hat{c},x,y)} = \begin{cases} \delta_{c,\hat{c}} \delta_{i,x} \delta_{j,y} & \text{if } I(\hat{c},x,y) > 0 \\ 0 & \text{otherwise} \end{cases}$$

and

$$\frac{\partial Z_s(c,i,j)}{\partial I(c,x,y)} = Z_s(c,x,y)(Z_s(c,x,y) - 1) \delta_{c,\hat{c}} \delta_{i,x} \delta_{j,y}.$$

III. Analysis

Different analysis are carried out on the model described in Table I to measure the efficiency of our implementation. A GPU has been used. First, the training time of our model is compared to the time taken by the PyTorch implementation for a similar model. The Batch size parameter n can be specified in the function `train` of our model. It corresponds to the number of samples passed through the model at once. The comparison of the two implementation as a function of n is visible in Figure 4.

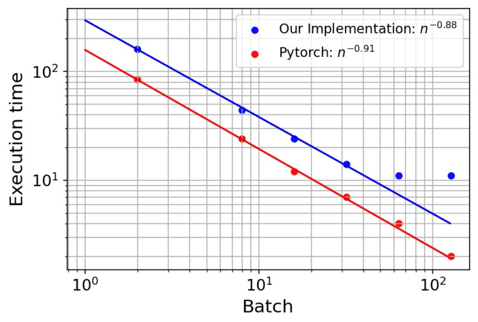

Fig. 4: Comparison of execution time as a function of the batch size for 11000 images, 10 epochs and $C = 16$.

Our implementation is nearly twice slower than the PyTorch implementation. In both, the higher the batch size, the smaller the training time. For our implementation, the training time seems to stabilize from a batch size of 64. Before that, the fit shows a proportional relation between the time and $1/n^{0.88}$.

The network and the number of training samples being small, using Stochastic Gradient Descent ($n = 1$) gives a reasonable training time, compared to the Pytorch implementation. However, with a bigger network, the batch size can really make a time difference and must not be neglected.

Moreover, the batch size also has an influence on the efficiency of the model. Indeed, a batch size being too large will lead to a poor generalization of the gradient and a batch size being too small has less probability to converge to a global optimum.

Figure 5 shows how well the model denoises the images during the training part. After each epoch and for different batch sizes, the Peak Signal-to-Noise Ratio (PSNR) is measured on the validation data set.

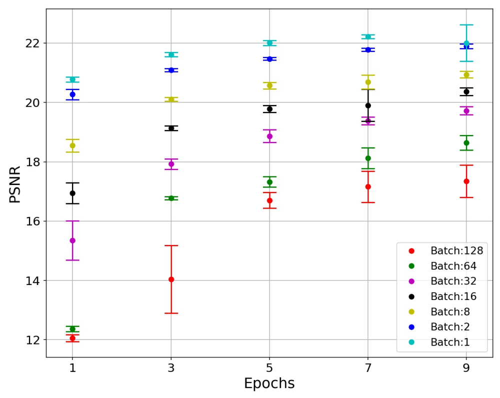

Fig. 5: PSNR value for different batch size during the training for 7700 images.

Being limited to 10 minutes with GPU, it is clear that a small batch size gives better results. After 10 epochs, the PSNR reaches nearly 22dB for batch sizes 1 and 2. The training with a batch size of 2 being nearly four times faster than with a batch size of 1, this value is kept for the next part.

The next step consists in studying the influence of the filter size k on the final result. The result is visible in Figure 6.

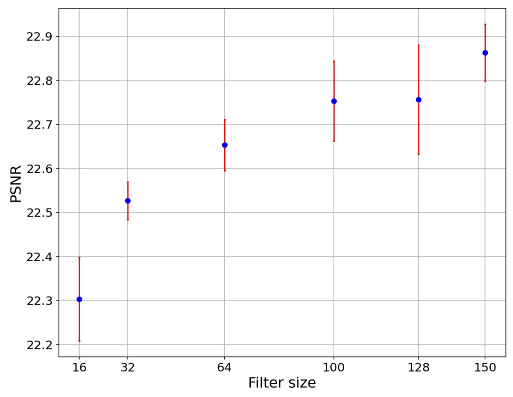

Fig. 6: PSNR value after 6 minutes of training for different filter size and 19000 images.

The bigger the filter size is, the higher the PSNR result is. The highest score obtained by training 10 minutes with the GPU and $k = 100$ is 23.1 dB. A denoised image, which was obtained by our best implementation, is compared to the groundtruth, the noisy image, and the denoised image computed by the usual Pytorch implementation (trained 10 minutes with $k = 100$) in figure 7. The denoised images present similar quality, which is represented by the mean PSNR (computed on a test set of 1000 images).

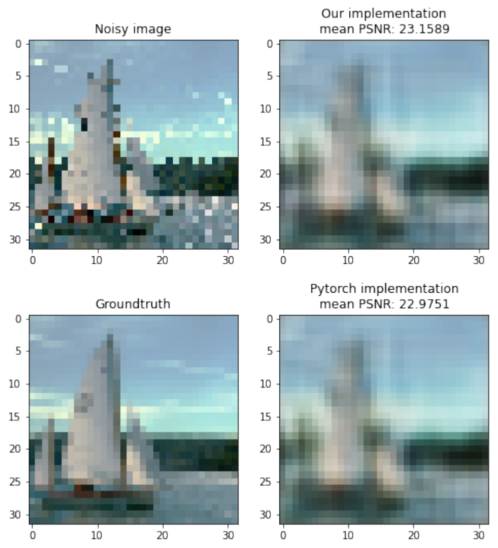

Fig. 7: Comparison of the denoised image obtained by our implementation (top right), the denoised image obtained by the usual pytorch implementation (down right), the groundtruth (down left) and the noisy image (up left).

IV. Discussion

In this project, the mathematics behind some of the most used modules of the library Pytorch have been studied and the said modules have been implemented with only the basic Python libraries. The speed advantage of matrix products over loops in python has been exploited, as well as some useful functions like `fold` and `unfold`, in order to implement the convolutional layer.

Then a study of the quality and the speed of the implementation with respect to the batch size was conducted. We concluded that the execution time was longer for smaller batch sizes, but the PSNR (i.e. the quality) of the denoised images was better over the epochs.

To conclude, the quality of the denoised images obtained by our final implementation is as good as pytorch implementation's. However, our implementation of a simple denoising model is slower than Pytorch's one. There is definitely room for improvement to reach the speed of the Pytorch implementation. Nevertheless, it shows how optimized the Pytorch implementation is, and how well it uses the GPU structure. Furthermore, a better score would certainly have been obtained with more training and more layers. The model implemented in the project was very simple, and other more efficient models have been found in mini-project 1 and could be employed. Implementing the better models would have required other modules and functions to implement and to optimize. For example, the pooling layers are used a lot for image denoising and implementing it would certainly have helped us perform a better score.

References

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[2] Jaakko Lehtinen, Jacob Munkberg, Jon Hasselgren, Samuli Laine, Tero Karras, Miika Aittala, and Timo Aila. Noise2noise: Learning image restoration without clean data. *arXiv preprint arXiv:1803.04189*, 2018.

[3] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.