Implementation of Noise2Noise models to learn image restoration without clean data

Miniproject 2

Camillo Nicolò De Sabbata, Stefan Igescu, Giovanni Monea
Department of Computer Science, EPFL, Switzerland

***Abstract*—This report presents our native implementation of a deep neural network that allows to restore images by only looking at other corrupted examples, without using clean data. Our implementation aims at building from scratch PyTorch modules for deep neural networks.**

I. Introduction

In this project, the goal is to build a framework for denoising images without using autograd or torch.nn modules. In particular we had to implement the following architecture, by building it from scratch:

- Conv(stride 2)
- ReLU
- Conv(stride 2)
- ReLU
- Upsampling
- ReLU
- Upsampling
- Sigmoid

The training set and the validation set are the same of before. Again, the performance metric used was the Peak Signal-to-Noise Ratio (PSNR).

II. Methodology

A. Implementation of PyTorch modules

Each module in the architecture is build as a class having the following methods:

- forward: implementing the forward pass
- backward: implementing the backward pass
- param: returning the parameters of the module, if any.
- zero_grad: which clears the gradients of the parameters of the module, if any.

Each module saves internally the value of the input during the forward pass and uses it for the backward pass.

1) Convolution: The parameters of the convolution module are the following:

- weight: representing the kernel of the convolution without the bias
- bias: representing the values of the bias of the kernel
- dl_dw: representing the gradient with respect to the weights
- dl_db: representing the gradient with respect to the bias.

The parameters are initialized using the Xavier method, to compensate for both the forward and backward passes.

In theory the forward pass of a convolution involves the sliding of the kernel over the input, the point-wise product of the two at each position and the sum of the result. However, in order to avoid for-loops in the implementation, we implemented the forward pass as a matrix product between the unfolded version of the input and the kernel. Once we obtain the result of the matrix product we fold it again in order to obtain the output of the convolution in the correct shape.

The backward pass has also been implemented as a convolution. Given that x is the input of the Convolution and that dl_out is the gradient of the output therefore we have:

- dl_dw: computing this is equivalent to computing the output of a classic convolutional layer which takes as input x and as kernel dl_dout. Whenever $stride > 1$, it is necessary to add a number of zeros equal to $stride - 1$ between the values of the kernel dl_dout in the same channel.
- dl_db: this is the result of the sum of the values in dl_dout for each channel. For example, dl_db of the first channel corresponds to summing the values in the first channel of dl_dout.
- dl_dx: this is the result of the convolution between dl_dout (with padding equal to the difference between the shapes of dl_dout and x, so that dl_dout has same shape as x) and the kernel rotated of 180°. It can happen that the kernel does not cover the entire input for particular combinations of padding, input size and kernel size (e.g: x_shape = (1,3,5,5), kernel_shape = (2,3,2,2), stride = 2, padding = 0). As a consequence, if we apply the aforementioned convolution, then the shape of the dl_dx we obtain corresponds only to the shape of the covered part of the input. Therefore we add zero values to dl_dx corresponding to the values that did not participate to the forward pass. Again, whenever $stride > 1$, before it is padded it is necessary to add a number of zeros equal to $stride - 1$ between the values of the kernel dl_dout in the same channel.

The convolution module accepts as parameters in_channels (number of channels of the input), out_channels (number of channels of the output), kernel_size, padding (default: 0), stride (default: 1), bias (default: True).

The method zero_grad fills dl_dw and dl_db with zeros.

2) Nearest neighbor upsampling: The only hyper-parameter of this module is the scaling factor.

The forward pass operates by replacing each value by a $scaling\_factor^2$ matrix with the corresponding value repeated. This is obtained with a particular reshaping and repetition of the matrix.
In the backward pass we compute only dl_dx since this module does not have parameters. In particular for each value in the input we compute its gradient by summing the values of dl_dout corresponding to the repeated values computed in the forward pass.

3) Upsampling: This module has not been implemented as a transposed convolution, but instead as a combination of a nearest neighbor upsampling and a convolution.
The forward pass is implemented by first applying the forward pass of nearest neighbour upsampling and afterwards the forward of the convolution.
In the backward pass instead we first apply the backward of the convolution and then the backward of the upsampling.

4) ReLU and Sigmoid: We implemented ReLU and Sigmoid and they have no particular internal parameters. In the forward pass they implement respectively the following functions:

$$g(x) = \max(0, x) \tag{1}$$

$$g(x) = \frac{e^x}{1 + e^x} \tag{2}$$

In the backward pass we implemented respectively the following functions:

$$g'(x) = \begin{cases} \frac{dl}{dout} & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

$$\sigma(x)(1 - \sigma(x))\frac{dl}{dout} \tag{4}$$

5) MSE: The MSE module computes the mean squared error in the forward pass as $\frac{1}{N}(pred - target)^2$, where N corresponds to $B * C * H * W$ (where B is the batch size, C the number of channels, H the height and W the width of the output) i.e the total number of data points in the output. The error $(pred - target)$ is saved internally and it is used to compute the result of the backward pass as $2\,(pred - target)$.

6) Sequential: This module wraps all the other modules of the architecture. It saves all the components in $self.modules$ which is simply a list of modules.
The forward pass is performed by traversing the list of modules and applying the forward method of each component and taking as input the result of the previous forward.
Instead, the backward pass traverses the list in the reversed order and applies the backward method of each component taking as input the result of the previous backward. In particular the first module that calls the backward takes as input the backward result of the MSE module.
De facto, the param method returns the parameters of only the convolution and upsampling modules (and, in particular, of its convolutional layer), since they are the only modules with parameters.
The zero_grad calls zero_grad of each component in the list $self.modules$.

7) SGD: This module implements the SGD optimizer as it is built in PyTorch. We also decided to include as parameters momentum, Nesterov's momentum, dampening and weight decay, even if the last two are not used in our implementation. The step method updates the parameters that needs to be learned in all the layers of our architecture by subtracting their gradient computed in the backward pass. In particular, in our architecture SGD takes the parameters from Sequential.
It also has the method zero_grad which fills with 0 the gradient of all parameters.

B. Hyperparameter tuning

We decided to perform a grid-search for the choice of the hyperparameters of the network.

In particular, we tried to use SGD with and without momentum, with and without Nesterov momentum and with three possible learning rates (LR), selected after an initial search of stable values by hand.

With respect to the convolutional layers, it is possible to change the kernel size for both down-sampling (DSKS) and up-sampling (USKS) layers (with a subsequent proper padding), as well as the number of "shallow" (i.e., after the first down-sampling) and "deep" (i.e., after the second down-sampling) hidden layers.

Finally, we tried different values of the batch size (BS) during training.

An extensive description of the possible hyperparameter values is shown in table I.

The total number of resulting possible models is 576. As the training of so many different models is quite heavy, we decided to train these models on 1000 training samples for 10 epochs.

Afterwards, we further trained the top 20 performing models with 5000 training samples for 20 epochs.

Finally, we selected the top performing model among these twenty and we trained it with the complete dataset for 5 epochs. However, even if the PSNR score has a satisfying value of 23.37, the high number of hyperparameters (due to the high number of hidden layers) caused a rather slow training. For this reason, we chose to train our best model with respectively 16 and 32 hidden layers among the top twenty.

Context	Hyperparameter	Possible values
SGD	Nesterov	True, False
SGD	Momentum	0, 0.9
SGD	Learning rate	0.001, 0.01, 0.05
Convolutions	Hidden layers	(8, 32), (16, 32), (16, 64), (32, 64)
Convolutions	Up-sampling kernel size	3, 5
Convolutions	Down-sampling kernel size	2, 4
Training	Batch size	4, 16, 64

TABLE I
SUMMARY OF HYPERPARAMETER VALUES

We show in table II the PSNR score for the top 20 models.

From the results, we can observe that, most times, a non-zero momentum and a learning rate of 10^{-2} are preferred. Moreover, the best batch size is 4 in almost all cases.

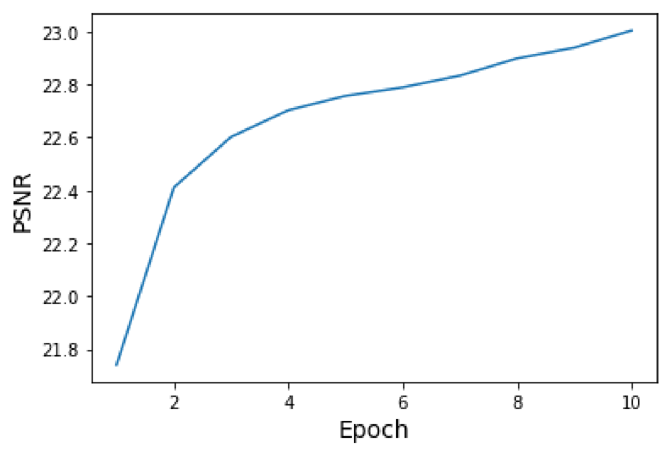

Fig. 1. Evolution of PSNR during training of our best model

Regarding the number of layers, we can see that larger layers perform better than smaller ones.

Nesterov	Momentum	LR	Layers	DSKS	USKS	BS	PSNR
True	0.9	0.01	(32, 64)	5	2	4	23.07
True	0.9	0.01	(32, 64)	4	3	4	22.94
True	0.9	0.01	(32, 64)	4	5	4	22.85
False	0.9	0.01	(16, 64)	2	5	4	22.79
True	0.9	0.01	(16, 64)	2	5	4	22.71
True	0.9	0.01	(16, 64)	4	5	4	22.65
True	0.9	0.01	(16, 64)	4	3	4	22.61
False	0.9	0.01	(32, 64)	4	3	4	22.61
True	0.9	0.01	(32, 64)	2	3	4	22.52
True	0.9	0.01	(16, 64)	2	3	4	22.44
False	0.9	0.01	(16, 64)	4	3	4	22.40
False	0.9	0.05	(32, 64)	2	3	16	22.40
True	0.9	0.01	(16, 32)	2	5	4	22.30
True	0.9	0.01	(16, 32)	2	3	4	22.18
False	0.9	0.01	(32, 64)	2	3	4	22.13
False	0.9	0.01	(16, 32)	2	3	4	21.94
False	0.9	0.01	(16, 64)	2	3	4	21.67
True	0.9	0.01	(8, 32)	2	3	4	21.38
True	0.9	0.05	(32, 64)	2	3	16	21.10
False	0.9	0.01	(8, 32)	2	3	4	20.76

TABLE II
20 TOP PERFORMING MODELS TRAINED FOR 20 EPOCHS WITH 5000 TRAINING SAMPLES

III. Results and discussion

Our final model resulted from the following assignment of the hyperparameters:

- Nesterov: True
- Momentum: 0.9
- Learning rate: 0.01
- Hidden layers: (16, 32)
- Down-sampling kernel size: 2
- Up-sampling kernel size: 5
- Batch size: 4

We trained it for 10 epochs with the full dataset. The final PSNR score of our model is **23.00**. The growth of the PSNR can be observed in Fig. 1

However, we observed that models with bigger hidden layers achieve better results, as they should allow for a richer and more complex latent space, even if their training is much slower.