# Noise2Noise from scratch : Miniproject 2 for Deep Learning (EE-559)

Elia Fantini, Kaan Okumuş, Kieran Vaudaux
*EPFL, Switzerland*
*May 27, 2022*

***Abstract*—Noise2Noise paper [2] from 2018 applied a Convolutional Neural Network to learn how to restore images and remove noise by training it using only noisy pictures. Applying basic statistical reasoning to signal reconstruction, they managed to achieve performances from similar to better of previous methods, which used clean targets for the supervised training. In this project we applied the same principles on a dataset of $50000$ pairs of $3$ channels $32 \times 32$ images, corrupted by two different noises. However, we will apply these principles using our own Pytorch-like framework, which we have implemented from scratch. Using a fairly simple network architecture and using the Adam optimizer, we achieved a PSNR of 22.6 dB for our best network architecture.**

## I. Introduction

The procedure of removing noise to improve the quality of an image, also called denoising, has been applied since many years. With the rise in popularity of deep networks and in particular with Convolutional Neural Networks (CNN), results on denoised images improved significantly. On the other hand, CNN require a lot of data to be trained and obtaining pairs of noisy images with the corresponding clean ground truth is not so easy in all applications. Noise2Noise paper [2] proposed a solution to this problem by using noisy images as the targets of training. Their intuition was to use a property of L2 loss minimization: on expectation, the estimate remains unchanged if we replace the targets with random numbers whose expectations match the targets. In other words, corrupting the training targets of a neural network with zero-mean noise will not change what the network learns. As for the first project, we were given a dataset of 50000 pairs of images, which have been downsampled to 3 channels $32 \times 32$ images. In every pair, the first picture is corrupted by the same but unknown noise, while the second picture is corrupted as well, but with a different noise. We were also given additional $1000$ pairs of noisy images with the corresponding clean targets to validate our training. Performance was measured using the peak signal-to-noise ratio (PSNR), a ratio between the maximum possible power of a signal and the power of corrupting noise that affects the fidelity of its representation. It was calculated with the following equation:

$$PSNR = 20 \log_{10} MAX_I - 10 \log_{10} MSE$$

$MAX_I$ is the maximum possible pixel value of the image, whereas $MSE$ is the mean square loss (L2). In our case $MAX_I$ was 1, since images were scaled in the range $[0, 1]$ before being used for training.

## II. Framework implementation, data pre-processing and model selection

### A. Deep Learning framework from scratch

The objective of this second project is to implement from scratch a framework similar to that of Pytorch, to be able to reproduce, to some extent, the experiments that were done in the first project. This required the implementation of several Modules, which were necessary to denoise images using Convolutional Neural Networks. The network used in the Noise2Noise paper, [2], uses mainly Convolutions and Maxpooling layers, as well as Transposed Convolutions layers as Upsampling layers. However, for the sake of simplicity we have chosen to use only Convolutional layers as well as Upsampling layers obtained by the combination of Nearest Neighbor upsampling + Convolution. The optimizer Stochastic Gradient Descent (SGD) not being able to reach quickly the desired convergence, we were inspired by the choice of the Noise2Noise paper and implemented the Adam algorithm, [1], in order to obtain a better convergence. Although, as discussed in the Noise2Noise paper, the use of L1 loss can be justified in some cases, but according to the results obtained in the first mini-project, this did not seem to have any added value in our case. Thus, as we seek to maximise the PSNR which is a decreasing function of the MSE, we have limited ourselves to the implementation and use of the MSE loss function.

### B. Data pre-processing

Although data normalisation and data augmentation generally allow for faster learning and better generalisation respectively, given the little improvement that these pre-processing steps brought in the first project, we chose to do without them in this project. In the following, we only re-scaled our data to have pixel values between 0 and 1.

### C. Model selection

The network, UNet, used in the Noise2Noise paper was relatively large, and was used to denoise images whose resolution was much higher than that of our $32 \times 32$ images. Since the framework in which the selected model will be trained is less optimised than the Pytorch environment, we have chosen not to reproduce this network, but rather to use the skeleton network proposed for this project.

The architecture used in the Noise2Noise paper significantly increases the depth along the dimension of the channels, however after doing some tests, it turned out that for the type of architecture we were considering, this did not allow for any real improvement in the training procedure and on the contrary slowed down our computations greatly. We therefore found a compromise between the depth of the network and the computation time, by setting the maximum depth to 12 channels. This was all the more justified as we were training our model on much smaller images than those in the paper. The Table I shows the efficiency of 4 networks inspired by the proposed architecture of the project, the details of each of these architectures can be found in the Appendices.

TABLE I
PSNR AND TEST LOSSES AFTER 3 EPOCHS FOR 4 DIFFERENT CHOICES OF NETWORK ARCHITECTURE

| Network | PSNR (dB) | Test Loss |
|---|---|---|
| Net1 | 21.92 | 0.0074 |
| Net2 | 22.058 | 0.0072 |
| Net3 | 19.76 | 0.012 |
| Net4 | **22.24** | 0.0069 |

These results were obtained using a batch size of 100, and the Adam optimizer with a learning rate of 0.001 and parameters $(\beta_1, \beta_2) = (0.9, 0.999)$. These results suggest that the Net4 network architecture is the most efficient for our denoising task. For the remainder of this project we will therefore base ourselves on this architecture.

## III. Optimization of the parameters

Now that the architecture of our network is fixed, we will try to optimize our hyperparameters, such as the batch size, the learning rate, $\beta_1$ and $\beta_2$, so that we can reach the best possible PSNR afterwards.

### *A. Batch size selection*

The batch size being an element that influences the learning speed as well as the computation time, we carried out some tests with different batch sizes, in order to find the one that would allow us to obtain the best compromise between convergence and computation time.This test was again done using the Adam optimizer with its default settings ($lr = 0.001$, $(\beta_1, \beta_2) = (0.9, 0.999)$). According to Table II, the best convergence is reached for a batch size of 400, but we can also see that batch sizes between 50 and 200 also allow a good convergence. Moreover, the computation time for the 3 epochs are globally identical for all batch sizes, except for the batch size of 1 which has an execution time almost twice as long as the others. This taken into account, it seems logical to us to continue by fixing the batch size at 400.

TABLE II
PSNR AND TEST LOSSES AFTER 3 EPOCHS FOR 6 DIFFERENT BATCH SIZE

| Batch size | PSNR (dB) | Test Loss | Computation times (s) |
|---|---|---|---|
| 1 | 22.2886 | 0.00691 | 1743.48 |
| 10 | 22.4816 | 0.00669 | 1408.66 |
| 50 | 22.5281 | 0.00665 | 541.32 |
| 100 | 22.5321 | 0.00663 | 604.10 |
| 200 | 22.5335 | 0.00663 | 627.09 |
| 400 | **22.5404** | 0.00660 | 568.40 |

### *B. Grid search for $\beta_1$ and $\beta_2$*

The parameters $\beta_1$ and $\beta_2$ of the Adam algorithm also have an important role in the convergence of our model. The parameters used in the Noise2Noise paper are the default parameters proposed by Pytorch, namely $\beta_1 = 0.9$ and $\beta_2 = 0.999$. During our tests, high $\beta_1$ and $\beta_2$ values (close to 1) generally allowed us to have better convergence and more stability throughout the training of the model. This is why we performed a grid search for these two parameters by setting the learning rate to 0.001 and by looking for optimal values for $\beta_1$ and $\beta_2$ close to 1. Table III shows the results of this grid search for $\beta_1 \in \{0.8, 0.85, 0.9, 0.99, 0.999\}$ and $\beta_2 \in \{0.8, 0.9, 0.99, 0.999\}$.

TABLE III
PSNR AND TEST LOSSES AFTER 3 EPOCHS FOR 6 DIFFERENT BATCH SIZE

| $\beta_1 \setminus \beta_2$ | 0.8 | 0.9 | 0.99 | **0.999** |
|---|---|---|---|---|
| 0.8 | 20.34 | 21.05 | 21.21 | 21.30 |
| 0.85 | 21.39 | 21.51 | 21.55 | 21.58 |
| **0.9** | 21.65 | 21.72 | 21.75 | **21.77** |
| 0.99 | 15.88 | 15.15 | 16.62 | 17.42 |
| 0.999 | 17.61 | 13.53 | 14.39 | 11.32 |

The optimal value of the parameters $\beta_1$ and $\beta_2$ seem to be the default values of Pytorch, namely $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

## IV. Learning rate

Another crucial parameter for the convergence of the Adam agorithm is the learning rate, indeed a high learning rate can allow to have good progress at the beginning, but it risks to make the algorithm unstable as one approaches the optimal value. Conversely, a low learning rate will make the algorithm more stable but it may also make convergence slower. In our tests we found that a leraning rate smaller than 0.001 made the convergence relatively slow and a value larger than 0.05 made the algorithm much too unstable. Figure 1 allows us to compare the behaviour of the algorithm for several learning rates. For a learning rate of 0.05, we can see that the convergence is unstable from the 5 epoch. For the other values of learning rate, their behaviour seems globally stable, however the learning rate 0.001, seems to offer a better convergence. Moreover, as we do not use any scheduler to decrease the learning rate during the training of the model, it seems judicious to us to fix the learning rate at 0.001 in order to guarantee a better stability for the continuation of the convergence.

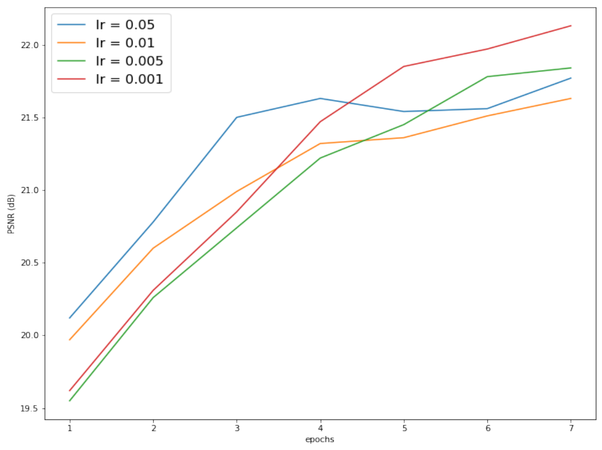

Fig. 1. PSNR achieved with 7 epochs for different value of the learning rate

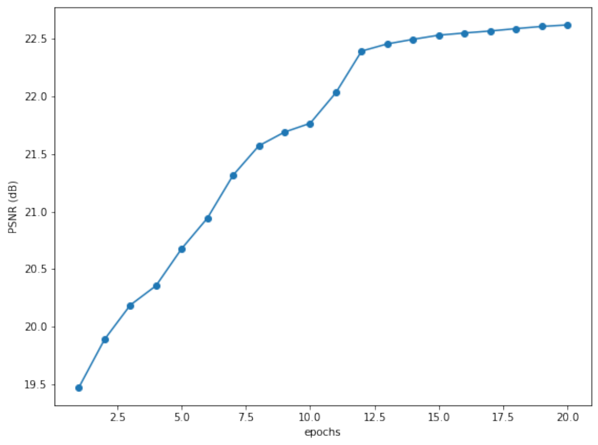

Fig. 2. Training of the final model.

## V. Results and Discussion

As can be seen in Figure 1, we managed to achieve a PSNR of 22.6 dB. Subsequently, we tried to improve this result by running further optimisations based on this "best model", but whether by increasing or decreasing the learning rate, or by varying the parameters $\beta_1$, $\beta_2$ and/or the batch size, we did not manage to obtain significantly better results. Nevertheless, as can be seen in Figure 3, the results of our denoising seem to be correct, even if we notice that our predictions remain blurrier than the target images.

The main objective of this project was the implementation of a from scratch framework to be able to reproduce, to some extent, the results we obtained in the first project. While the first project allowed us to achieve a PSNR of 25.6 dB using the Pytorch environment and a network with a fairly large architecture, this second project allowed us to achieve a PSNR of 22.6 dB with a fairly modest network size. The key factor that allowed us to achieve this result was the implementation of another optimizer than the SGD, namely the Adam optimizer. Subsequently, most of our efforts were focused on optimising the parameters of the Adam optimizer. This led us to obtain optimal values of 400 for the batch size, $(\beta_1, \beta_2) = (0.9, 0.999)$, as well as $0.001$ for the learning rate, these last three values being the default values attributed to these parameters in the majority of deep learning environments. This at least confirms that these parameters seem to be the most adequate for the training process of our model.

As a future improvement of this project, the implementation of a new Upsampling module using Transposed Convolution would be interesting to study in comparison with the one we have implemented which combines Nearest Neighbor upsampling + Convolution.

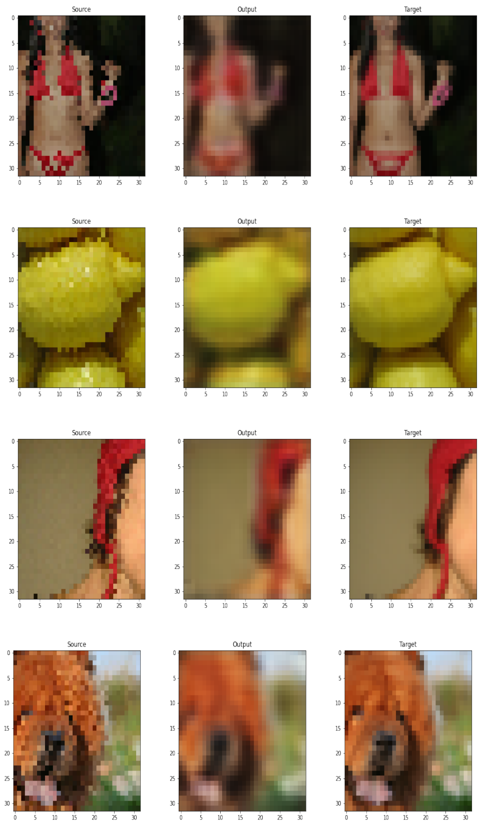

Fig. 3. Final model: example of noisy image (left), model's output (center), and clean target (right).

## References

[1] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: (2014). DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

[2] Jaakko Lehtinen et al. "Noise2Noise: Learning Image Restoration without Clean Data". In: *CoRR* abs/1803.04189 (2018). arXiv: 1803.04189. URL: http://arxiv.org/abs/1803.04189.

## APPENDIX A
## NET1 ARCHITECTURE

**Input**
$\downarrow$
**Conv2d(**$C_{in} = 3, C_{out} = 12, kernel = (4,4), stride = 2, padding = 2$**)**
$\downarrow$
**ReLU()** $\downarrow$
**Conv2d(**$C_{in} = 12, C_{out} = 9, kernel = (4,4), stride = 2, padding = 2$**)**
$\downarrow$
**ReLU()** $\downarrow$
**Upsampling(**$C_{in} = 9, C_{out} = 12, scale\_factor = 2, kernel = (4,4), padding = 1$**)**
$\downarrow$
**ReLU()** $\downarrow$
**Upsampling(**$C_{in} = 12, C_{out} = 3, scale\_factor = 2, kernel = (3,3)$**)**
$\downarrow$
**Sigmoid()** $\downarrow$
**Output**

## APPENDIX B
## NET2 ARCHITECTURE

**Input**
$\downarrow$
**Conv2d(**$C_{in} = 3, C_{out} = 12, kernel = (3,3), stride = 2, padding = 2$**)**
$\downarrow$
**ReLU()** $\downarrow$
**Conv2d(**$C_{in} = 12, C_{out} = 12, kernel = (3,3), stride = 2, padding = 2$**)**
$\downarrow$
**ReLU()** $\downarrow$
**Upsampling(**$C_{in} = 12, C_{out} = 9, scale\_factor = 2, kernel = (4,4)$**)**
$\downarrow$
**ReLU()** $\downarrow$
**Upsampling(**$C_{in} = 9, C_{out} = 3, scale\_factor = 2, kernel = (3,3)$**)**
$\downarrow$
**Sigmoid()** $\downarrow$
**Output**

## APPENDIX C
## NET3 ARCHITECTURE

**Input**
$\downarrow$
**Conv2d(**$C_{in} = 3, C_{out} = 12, kernel = (2,2), stride = 2$**)**
$\downarrow$
**ReLU()** $\downarrow$
**Conv2d(**$C_{in} = 12, C_{out} = 9, kernel = (2,2), stride = 2, padding = 1$**)**
$\downarrow$
**ReLU()** $\downarrow$
**Upsampling(**$C_{in} = 9, C_{out} = 12, scale\_factor = 3, kernel = (4,4), stride = 2$**)**
$\downarrow$
**ReLU()** $\downarrow$
**Upsampling(**$C_{in} = 12, C_{out} = 3, scale\_factor = 3, kernel = (5,5)$**)**
$\downarrow$
**Sigmoid()** $\downarrow$
**Output**

## APPENDIX D
## NET4 ARCHITECTURE

**Input**
$\downarrow$
**Conv2d(**$C_{in} = 3, C_{out} = 12, kernel = (2,2), stride = 2$**)**
$\downarrow$
**ReLU()** $\downarrow$
**Conv2d(**$C_{in} = 12, C_{out} = 9, kernel = (2,2), stride = 2, padding = 1$**)**
$\downarrow$
**ReLU()** $\downarrow$
**Upsampling(**$C_{in} = 9, C_{out} = 12, scale\_factor = 2, kernel = (2,2)$**)**
$\downarrow$
**ReLU()** $\downarrow$
**Upsampling(**$C_{in} = 12, C_{out} = 3, scale\_factor = 2, kernel = (3,3)$**)**
$\downarrow$
**Sigmoid()** $\downarrow$
**Output**