Noise2Noise - PyTorch Implementation

Gieruc Théo 282360
Krafft Guillaume 289256

ABSTRACT

This report presents the first part of the mini-project for the course EE-559 Deep Learning. In this part, our goal is to implement a self-supervised denoising neural network similar to what was done in the Noise2Noise paper [2].

1 INTRODUCTION

Our approach was divided in two parts: designing the network as described in the Noise2Noise paper, then tuning the hyperparameters in order to get better denoising results.

The network itself (as seen in Figure 1) is mostly composed of convolution layers as well as maxpool and upsampling layers. There are also four skip connections.

2 NETWORK DESIGN

2.1 Modular blocks approach

The network we implemented can be compartmentalized in smaller blocks to reduce the complexity of the code. We started with the creation of the class `ConvLeakyReLU` which is a convolution followed by a LeakyReLU activation function. This is useful, because all convolutions are followed by LeakyReLU activation, except for the output layer.

We created blocks that encapsulate elements such as ConvLeakyReLU, Maxpool or Convolutions. Those blocks are shown in Figure 2.

Those blocks are then placed one after the other in the `forward`, with an interpolation before each skip connections. This is shown in Figure 3.

We use a stack system in order to create the skip layers. We append the output of our `enc_conv` blocks on the stack, and pop them out of the stack to use them for concatenation in the decoder part of the network, shown by the horizontal arrows in Figure 3.

2.2 Training

For the training, the network takes batches of images with values between 0 and 1, so we divide the training images by 255. These images are stored into a custom dataset that concatenates them, applies transforms as explained in section 3.2 and gives them together for training.
We compared the Adam and SGD optimizer, and although we got faster convergence on Adam, the SGD has better stability and PSNR results. We decided to keep the SGD optimizer.

2.3 Prediction

For the prediction, we input our network with batches of images with values between 0 and 1. For the output, we first normalize the images. We tried several normalization methods, such as scaling each image so that it's minimum is 0 and its maximum is 1, doing the same but on the whole batch, or simply setting each value lower that 0 to 0, each value greater than 1 to 1. This last method gave the best results on PSNR. Finally, we multiply the batch by 255 so that the images are in the right format.

NAME	N_{out}	FUNCTION
INPUT	n	
ENC_CONV0	48	Convolution 3×3
ENC_CONV1	48	Convolution 3×3
POOL1	48	Maxpool 2×2
ENC_CONV2	48	Convolution 3×3
POOL2	48	Maxpool 2×2
ENC_CONV3	48	Convolution 3×3
POOL3	48	Maxpool 2×2
ENC_CONV4	48	Convolution 3×3
POOL4	48	Maxpool 2×2
ENC_CONV5	48	Convolution 3×3
POOL5	48	Maxpool 2×2
ENC_CONV6	48	Convolution 3×3
UPSAMPLE5	48	Upsample 2×2
CONCAT5	96	Concatenate output of POOL4
DEC_CONV5A	96	Convolution 3×3
DEC_CONV5B	96	Convolution 3×3
UPSAMPLE4	96	Upsample 2×2
CONCAT4	144	Concatenate output of POOL3
DEC_CONV4A	96	Convolution 3×3
DEC_CONV4B	96	Convolution 3×3
UPSAMPLE3	96	Upsample 2×2
CONCAT3	144	Concatenate output of POOL2
DEC_CONV3A	96	Convolution 3×3
DEC_CONV3B	96	Convolution 3×3
UPSAMPLE2	96	Upsample 2×2
CONCAT2	144	Concatenate output of POOL1
DEC_CONV2A	96	Convolution 3×3
DEC_CONV2B	96	Convolution 3×3
UPSAMPLE1	96	Upsample 2×2
CONCAT1	$96+n$	Concatenate INPUT
DEC_CONV1A	64	Convolution 3×3
DEC_CONV1B	32	Convolution 3×3
DEV_CONV1C	m	Convolution 3×3, linear act.

Figure 1: Noise2Noise network [2]

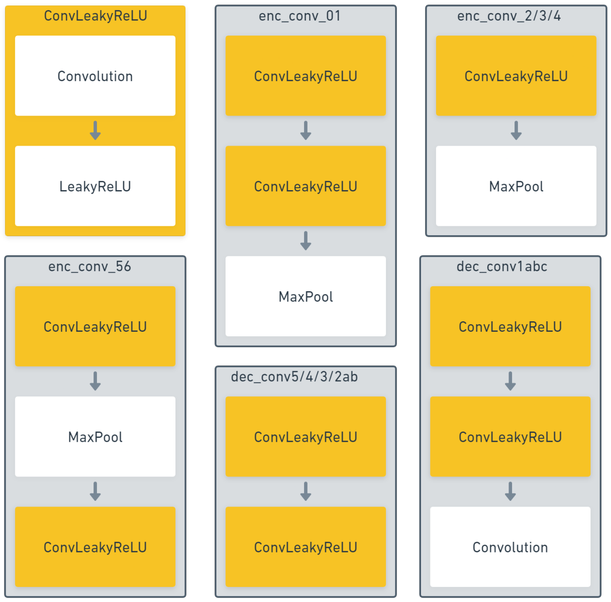

Figure 2: Basic blocks

3 TUNING

3.1 Hyperparameters tuning

In order to get better PSNR scores, we tried different modifications and tuning of hyperparameters, such as different activation functions, the introduction of batch normalization and learning rates of our optimizer.
We used Ray [1] with HyperOpt, a library for parallel optimization over a search space. The results are shown in Table 1. We concluded that the ReLU activation is not better than the LeakyReLU, that the use of Batch Normalization make the training 1.5 times slower while having slightly worse results. We noticed that a high learning rate and a small momentum works as well as a small learning rate and bigger momentum.

Loss	Activation	BN	Lr	Momentum	Time [s]
0.00375	LeakyRelu	FALSE	0.05521	0.18	3780
0.00378	ReLu	FALSE	0.00801	0.86	3796
0.00383	LeakyRelu	TRUE	0.03571	0.18	5306
0.00448	ReLu	FALSE	0.00130	0.24	2612
0.00458	ReLu	FALSE	0.00081	0.43	3774
0.00462	LeakyRelu	FALSE	0.00059	0.44	3779
0.00480	LeakyRelu	TRUE	0.00178	0.21	5345
0.00480	ReLu	TRUE	0.00404	0.31	5031
0.00494	LeakyRelu	FALSE	0.00018	0.66	3274
0.00605	ReLu	TRUE	0.00024	0.81	5227

Table 1: Tuning using Ray

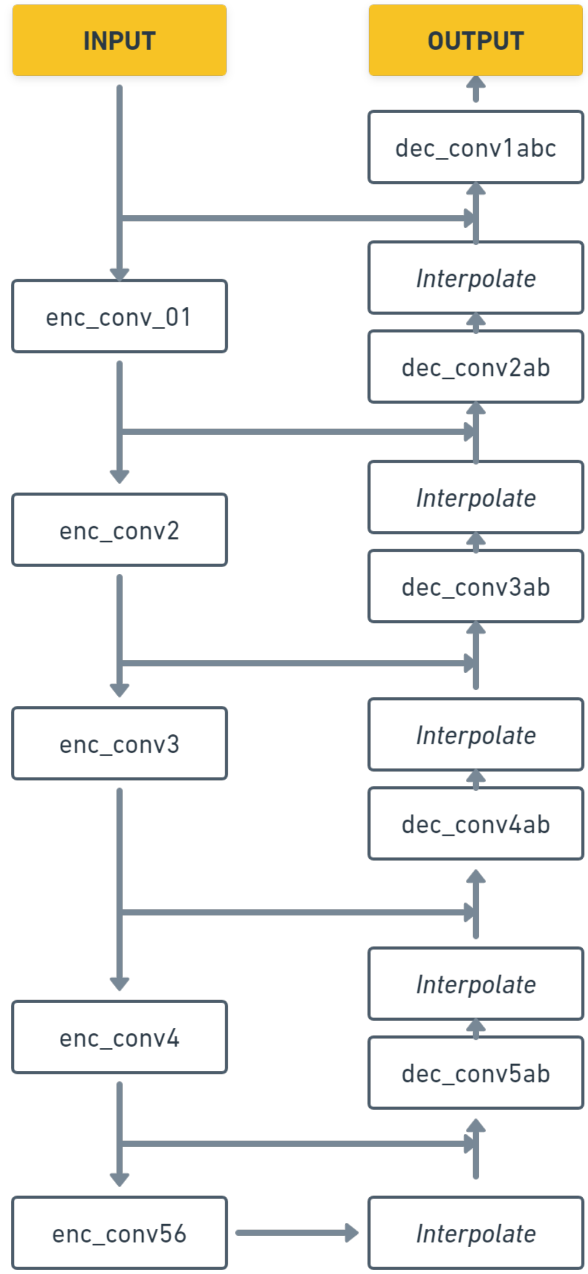

Figure 3: Forward function

3.2 Data augmentation

To improve the results of our network, we tried conducting data augmentation. We used the `torch.torchvision.transforms` library on ten epochs per training, averaging on ten training. We decided to try three augmentations: flipping horizontally and vertically, swapping the source and the target and changing the hue and brightness. The results are shown in Table 4 and an example of augmentation (flip and swap) is shown in Figure 4.

It is very important to use the same augmentation on both the source and the target, so we first concatenate both images, apply the augmentation and separate them for training. The results 4 show that the hue and brightness gives out worse result and is more computational intensive. We decided to keep the horizontal/vertical flip and the random swapping of target and source.

Figure 4: Example of augmentation

Augmentation	PSNR	loss	Time [s]
None	24.57	0.0144	760
flipVH	24.59	0.0145	767
flipVH + swapXY	24.63	0.0145	773
flipVH + swapXY + hue, brightness	24.56	0.0137	1500

Table 2: Our augmentation results

4 RESULTS

4.1 Learning time

On a mobile RTX 3060, a training over one epoch takes about 70 seconds.

4.2 Method

We first trained on a few epochs, with a PSNR of 24.5dB with a learning rate of 0.1 and momentum of 0.8. When then lowered the learning rate to 0.001 and did a few iterations, until reaching 25.51dB. We used a batch size of 8.
The loss for a training over a few epochs is shown in figure 6.
A few results are shown in Figure 5.

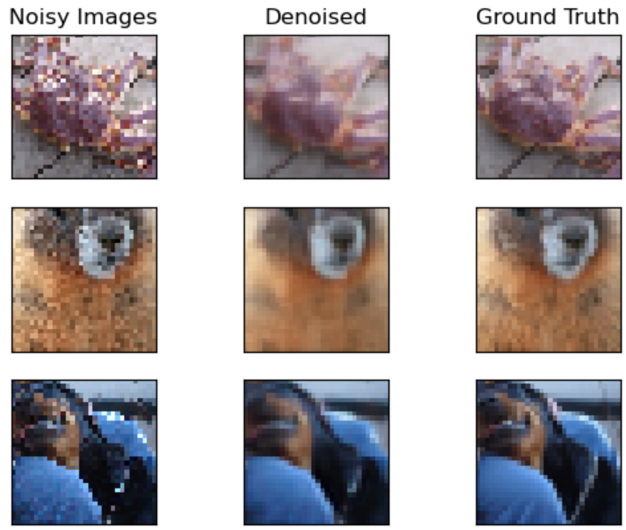

Figure 5: Noised, denoised and ground truth comparison

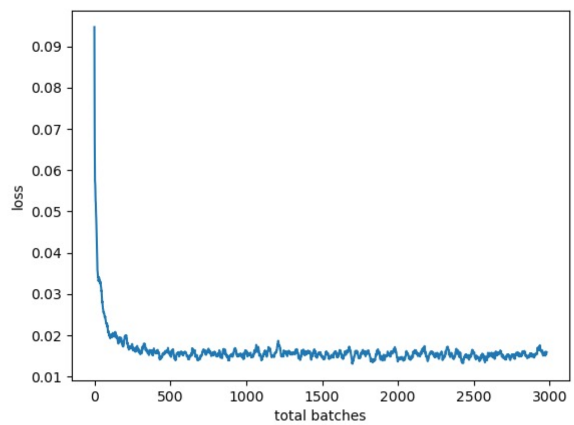

Figure 6: Loss according to number of batches

REFERENCES

[1] [n.d.]. Ray 1.12.1. https://docs.ray.io/en/latest/ray-overview/index.html
[2] Jaakko Lehtinen, Jacob Munkberg, Jon Hasselgren, Samuli Laine, Tero Karras, Miika Aittala, and Timo Aila. 2018. Noise2Noise: Learning Image Restoration without Clean Data. https://doi.org/10.48550/ARXIV.1803.04189