Mini-project 1: Image denoiser network using standard PyTorch framework

Mehrsa Pourya and Farzad Pourkamali
EE559 (Deep Learning) Course Project, EPFL

I. Introduction

In this mini-project, we implement an image denoiser network using PyTorch [1]. The used network architecture is a convolutional autoencoder that generates image-to-image mappings. The input and the target pairs of the training data are noisy images, and no clean reference is involved in the training process. This model is known as a Noise2Noise network [2], and there is statistical reasoning why it denoises images. The optimization task related to the training of a network f_θ with parameters θ, loss function L, and the noisy input-clean target pairs $(\hat{x}, y)$ is:

$$\arg\min_{\theta} \mathbb{E}_{(\hat{x},y)}\{L(f_\theta(\hat{x}), y)\}. \tag{1}$$

By using the conditional expectation (1) is formulated as:

$$\arg\min_{\theta} \mathbb{E}_{\hat{x}}\{\mathbb{E}_{(y|\hat{x})}\{L(f_\theta(\hat{x}), y)\}\}. \tag{2}$$

If we assume that the input samples are completely random, then the network tries to find the output from the distribution $p(y|\hat{x})$ with the minimum loss for each input. The distribution $p(y|\hat{x})$ may include many possible solutions with the same minimum loss. If the loss is the mean square error, the network outputs a solution that is the average of all valid outputs for a given input. Hence, one can corrupt the target values of a training set and get solutions that convey the lowpass components of all possible outputs. This observation is well suited to the denoising task. Consequently, one can train a network with corrupted inputs $\hat{x}$ and corrupted outputs $\hat{y}$ under the constraint that $\mathbb{E}\{\hat{y}|\hat{x}\} = y$ and still get the desired output y for the input $\hat{x}$ given that sufficient data is provided.

We use a training dataset that includes 50000 pairs of noisy input-target samples. Each sample is a 32×32 RGB image. The noise here stems from the downsampling and pixelating effects, and the goal is to denoise images to reduce those effects. An example of input-target pairs is illustrated in the first two columns of Fig. 4 in the Appendix.

We organize the report as follows: In Section II, we discuss the pre-processing and data augmentation, we explain the network architecture in section III, and we discuss parameters involved in the training process in Section IV. Finally, in Section V, we present our results in terms of the PSNR [1] metric on the provided validation data.

[1]Peak Signal to Noise Ratio

II. Pre-processing and data augmentation

As a pre-processing step, we cast the data type of input and target tensors to float and normalize the pixel values between $[0, 1]$ by dividing by 255. To construct a dataset from the input-target tensors, we write a class `MyDataset` that inherits the `torch.utils.data.Dataset` class. We rewrite the `__getitem__` method of this class to be able to use the transforms of torchvision for data augmentation. The noticeable fact here is that we should apply the same transform to the input and target pairs. Hence, we use the functional transforms instead of the random transforms of torchvision [3] which are not reproducible for two images. We control the randomness of the scheme by generating a random number and putting a threshold on its value. We try transformations like vertical and horizontal flips, brightness adjustment, rotation, and gaussian blurring. Using the constructed dataset, we instantiate an `torch.utils.data.DataLoader` object. The `Dataloader` also gets a mini-batch size as an input and generates batches of the training data with a random shuffling in each epoch. Each sample in a batch is either the original or a transformed image with a probability controlled by the mentioned random number. The selected transformations at the end are flipping and brightness adjustment. As we see in Table VI, data augmentation results in a subtle increase in validation PSNR.

III. Network architecture

The architectures used for image-to-image tasks are, in general, convolutional auto-encoders. Such networks map the input images to a low dimensional latent space by performing convolutions, downsampling, and increasing the number of channels (encoder). The downsampling can also be modeled as the stride parameter of the convolution layers. Then, the latent space is mapped back to the image space using a decoder network. The decoder includes convolutional and up-sampling layers. Equivalently, transpose convolutions with arbitrary strides are used in the decoder. In this mini-project, we try two different architectures: first, an autoencoder with ten convolutional layers. Second, we use an architecture similar to the Unet used in the Noise2Noise paper (Fig. 3) with some modifications. In both cases, we add a sigmoid activation to the output layer to ensure that the output values of the networks are in the range $[0, 1]$, then we multiply the output by 255 to be consistent with the project instructions.

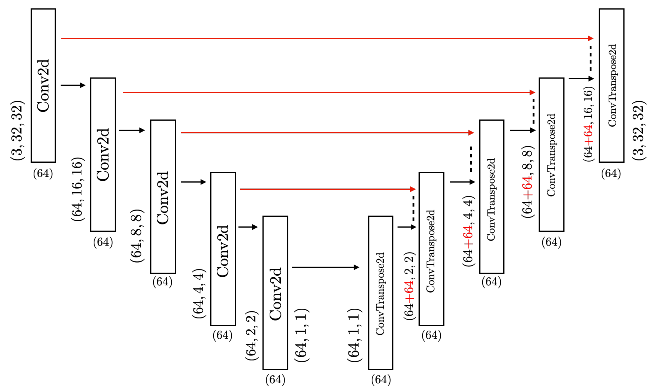

Fig. 1: The first proposed architecture with ten convolutional layers: each block represents a convolutional layer, red lines are skip connections and dashed lines represent concatenations. The size of images are written as (N-channel, height, width) in the input and output of layers.

A. Ten Conv-layer Autoencoder

We use five convolutional layers for the encoder with a kernel size of two and stride of two to map the 32×32 images to 1×1 dimensional images. We use the same structure for the decoder with transpose convolutions replacing the conventional convolutions. The number of output channels of each convolutional layer is 64 except for the last layer, which is 3. The activation functions between the layers are ReLU functions. However, when we train the model, we see that it does not achieve a good validation PSNR. Based on the experiments in min-project 2, a smaller network with four layers achieves a good PSNR. Hence, the reason that the proposed architecture does not work is probably vanishing gradients. To overcome this issue, we have two ideas: first, adding batch normalization blocks between the convolutional layers, and second, adding skip connections. Such connections help the network to produce identity mappings and could result in a remarkable boost in the network's performance. We tried both approaches and observed that the network with the combination of skip connections and batch normalization reaches the best validation PSNR in this part. We also tried adding drop-out layers between convolutional layers. However, it did not improve the validation PSNR. We report the validation PSNR of each case in Table I in Section V. We show a block diagram of this architecture in Fig. 1.

B. Noise2Noise Unet with Modifications

Our second architecture is similar to the Unet used in the Noise2Noise paper (Fig. 3). This architecture is similar to the one designed in the previous part. Both are convolutional autoencoders with skip connections, but with differences mentioned below:

1) It has more convolutional layers without downsampling, which preserves the size of the image.
2) The kernel size is different.
3) Instead of using strides, the downsampling is done by MaxPool blocks, which replace a window of a given size with its maximum.
4) The activation functions are leaky ReLUs. The value of leaky ReLU is not zero for the negative inputs, and it is a linear piece with a slope of α for such inputs. In this project, $\alpha = 0.1$.
5) Instead of transpose convolution, a combination of convolution and nearest neighbor upsampling is used.

In addition, we add three fully connected (Linear followed by ReLU functions) in the latent space where the size of images is reduced to 1×1 for 32×32 input images. If the input images are not of the 32×32 size, the network will not function. As we see in Table II the results section, this architecture achieved the best-achieved validation PSNR.

IV. Training process: optimizer, loss, and scheduler

In this section, we introduce the different parameters that affect the learning process and mention our tried cases for each parameter.

A. Batch Size

Batch size is the number of samples passed to the model in one forward iteration. For the final model, we tried three different batch sizes: 8, 16, and 32. In general, smaller batch sizes result in better generalization. However, for our final model, as we see in table III, there is not a meaningful effect of batch size. Hence, we chose the batch size of 32 to increase the parallelism while preserving the same validation accuracy level.

B. Loss Functional

We tried three loss functionals: mean square error (MSE), mean absolute error (L1), and smooth L1 loss. The latter acts like MSE for points that are closer than a specified threshold and as L1 loss when they are further. As reported in Table V, smooth L1 loss is doing slightly better than MSE, but the difference is not considerable. For simplicity, we chose MSE as our final loss functional.

C. Optimizer

We tried three different optimizers: Stochastic gradient descent with a learning rate of 0.1 and momentum of 0.9, Adam optimizer with a learning rate 0f 0.001, and the Adagard with a learning rate of 0.001. As we see in Table V, the Adam optimizer performs better than the others and is our chosen optimizer.

D. Scheduler

This module helps control the learning rate of the optimizer in different epochs. We use a `MultiStepLR` scheduler that decreases the learning rate by a factor of gamma in chosen epochs. We selected gamma as 0.1 and added a scheduler to the training. Scheduler results in a better convergence, and as we see in table VI, it produces better validation PSNR:

V. Results

In this section, we discuss the results of described schemes. We introduce our best architecture and learning parameters. To have fair comparisons, the basic setup for the experiments is as follows: optimizer is Adam with a learning rate of 0.001, and the loss is MSE. The batch size is 16, and no scheduler or data augmentation is used. The architecture is also the Unet with added fully connected layers. We only mentioned the changes that we made to the basic setup in each experiment, and other parameters remain unchanged. For calculating PSNR, we use the function provided in `test.py`. The first experiment is to fix the architecture. We compared all proposed architectures and summarize the results in Tables I & II. Henceforth, We hence choose the **Unet with added fully connected** as our architecture.

TABLE I: Validation PSNR for the first proposed architecture: Ten Conv-Layer network (TC), with addition of batch normalization (BN), skip connections (S) and drop out (DP)

TC	TC + BN	TC + S	TC + S + BN	TC + S + BN + DP
19.48dB	20.24dB	24.65dB	**24.78**dB	23.48dB

TABLE II: Validation PSNR for the second proposed architecture: Unet and Unet with added fully connected (fc) layers

Unet	Unet + fc
25.49dB	**25.51**dB

Then, we then chose the batch size for the fixed architecture. Table III summarize the results for this experiment. We chose **batch size of 32** as discussed before.

TABLE III: Effect of batch size

Batch Size:	8	16	32
	25.54dB	25.51dB	25.53dB

From now on we use the batch size of 32. Table V showes the validation PSNR achieved with different optimizer. Hence **Adam with a learning rate of 0.001** is our chosen optimizer.

TABLE IV: Effect of optimizer

Optimizer :	SGD	Adam	Adagrad
	24.53dB	25.53dB	23.99dB

Table V demonstrate the results for different used loss functions. Our final chose is **MSE**.

TABLE V: Effect of Loss

Loss:	MSE	L1	SmoothL1
	25.53dB	24.69dB	25.55dB

Finally, we investigate the effect of **data augmentation** and the **scheduler**. The chosen epochs for the decrease of the learning rate of the scheduler are 5 and 8. Both approaches improve the performance and are included in the final model.

TABLE VI: Effect of Data Augmentation (D.A.) and Scheduler

Final Model	+ D.A.	+ D.A. + Scheduler
25.53dB	25.56dB	25.66dB

Best Model: To conclude, we run the Unet architectures with: Loss as MSE, Adam optimizer with learning rate of 0.001, batch size of 32, and data augmentation. We run 100 epochs. Our scheduler reduces the learning rate at 30th and 60th epochs by a factor of 0.1. The achieved validation PSNRs are reported in Table VII. Although the model with fc layers gets a better PSNR, but it only works for 32×32 images. So, we use the Unet in our provided codes. For the timing, one epoch of training takes about 30 seconds on NVIDIA GeForce RTX 3090 GPU and reaches a PSNR of 24.99dB.

TABLE VII: Final best models

Unet	Unet + fc
25.807dB	**25.810**dB

For the best model, we plot the convergence curve of the loss in Fig. 2 to ensure that the training is done correctly. Further more in Fig. 4 and Fig. 5 of the Appendix we bring examples of denoised images of training and validation data.

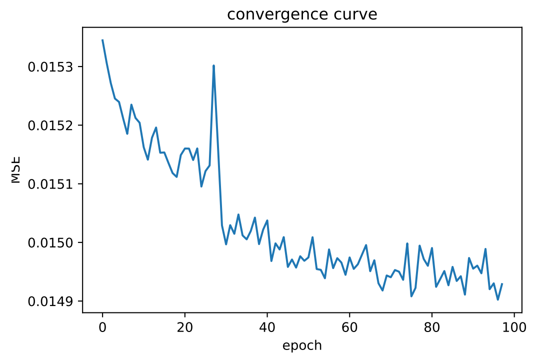

Fig. 2: Convergence curve for the training of Unet. Here we see the effect of the change of the learning rate in the 30-th epoch by scheduler.

References

[1] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[2] J. Lehtinen, J. Munkberg, J. Hasselgren, S. Laine, T. Karras, M. Aittala, and T. Aila, "Noise2noise: Learning image restoration without clean data," *arXiv preprint arXiv:1803.04189*, 2018.

[3] "Torchvision transforms," https://pytorch.org/vision/stable/transforms.html.

APPENDIX

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.

Fig. 3: The Unet architecture used in Noise2Noise paper, n and m are 3 in our project.

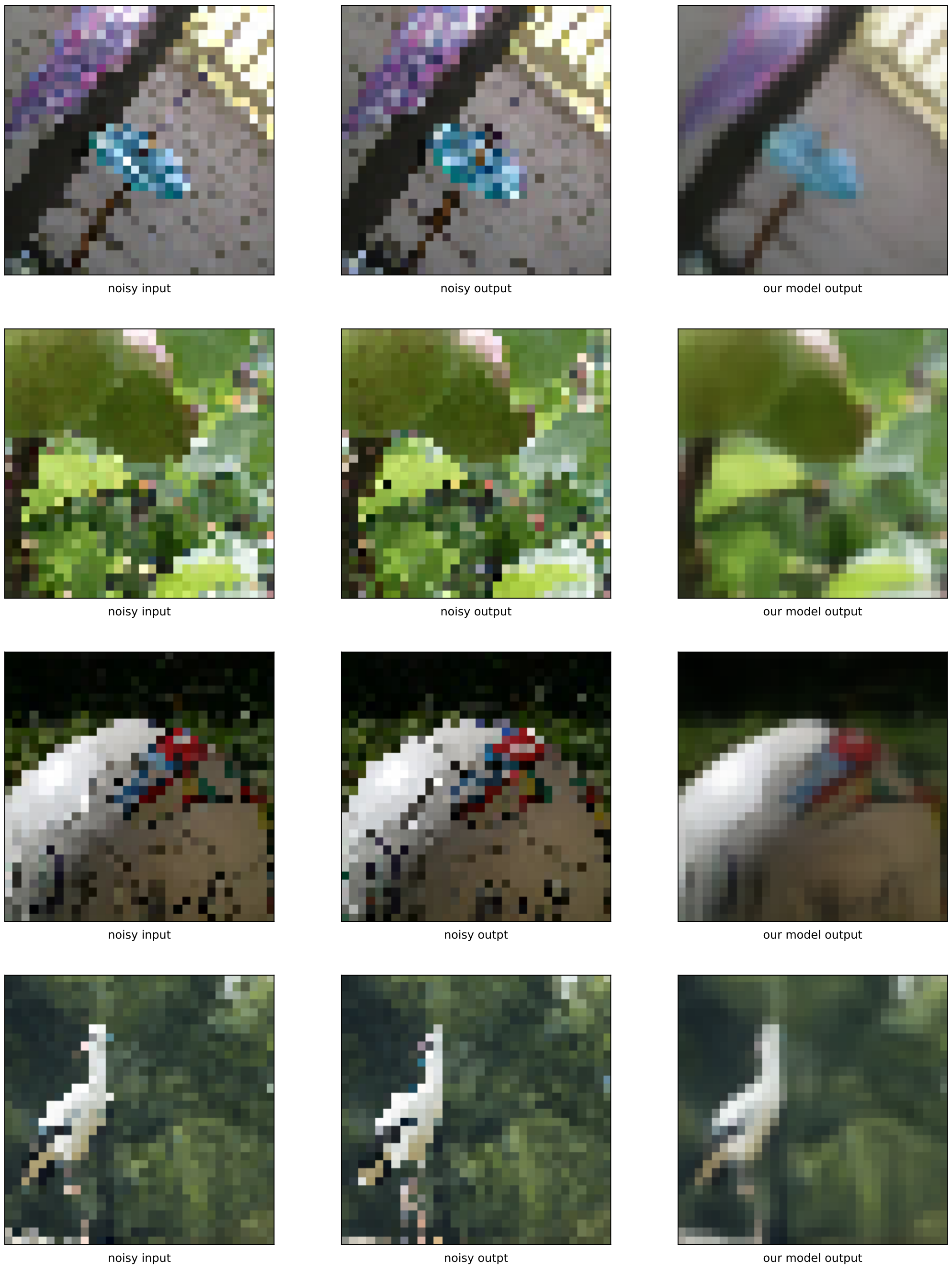

Fig. 4: Noisy input, noisy output and the output of our final model for the training data

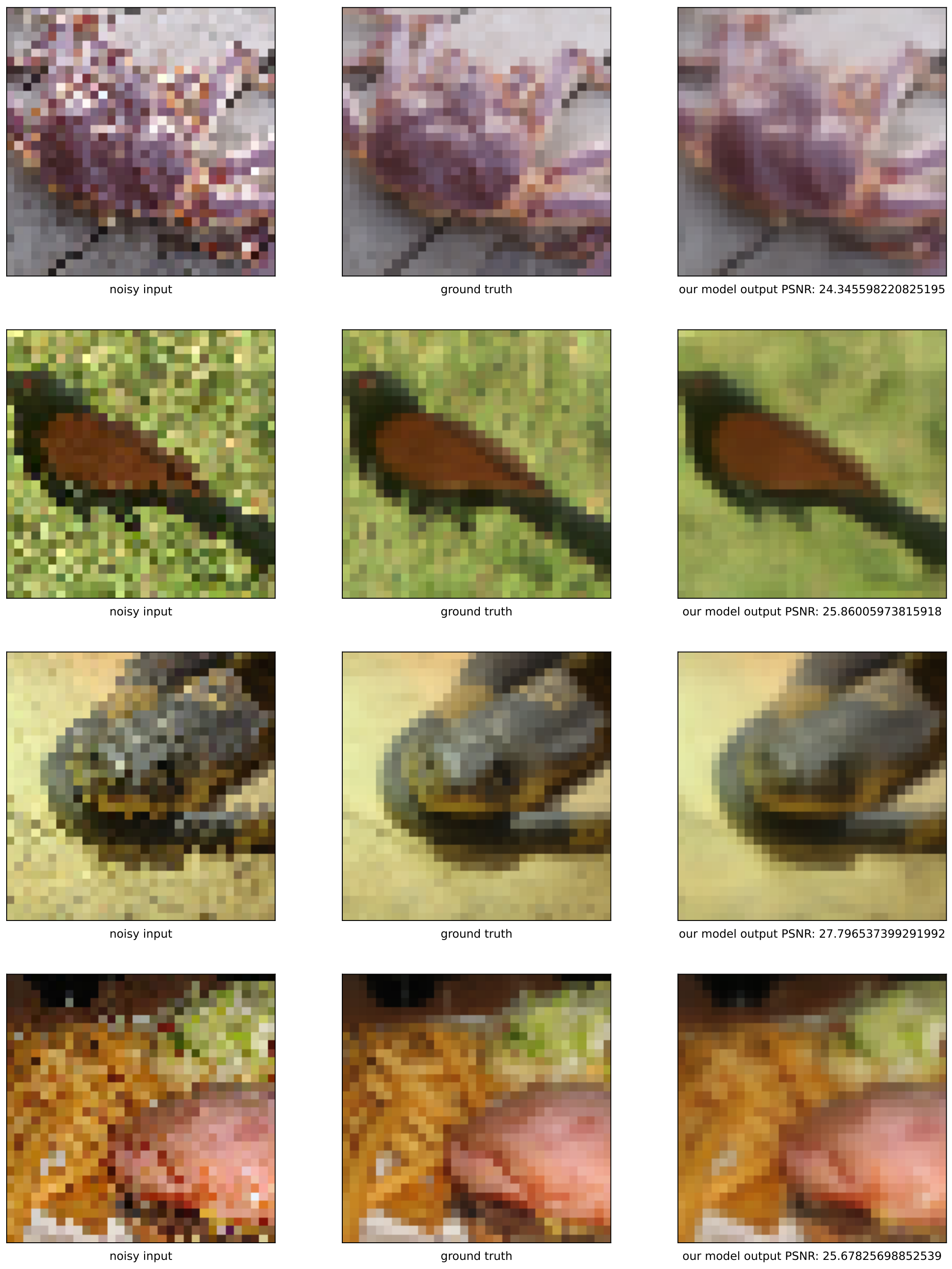
noisy input
ground truth
our model output PSNR: 24.345598220825195
noisy input
ground truth
our model output PSNR: 25.86005973815918
noisy input
ground truth
our model output PSNR: 27.796537399291992
noisy input
ground truth
our model output PSNR: 25.67825698852539

Fig. 5: Noisy input, noisy output and the output of our final model and its PSNR for the validation data