EE-559 Deep Learning

Mini-project 1: Noise2Noise auto-encoder using PyTorch Framework

Tom MERY **297217**

Spring 2022

Contents

List of Figures

1 Introduction

In this mini-project, a Noise2Noise model is implemented using the standard PyTorch Framework. In this report, the experiments and their results that have been carried out during the building process of the architecture are presented. The theoretical background about Noise2Noise auto-encoder is available in the original paper [1].

The provided dataset for training is composed of two tensors of the size $50000 \times 3 \times 32 \times 32$ that corresponds to 50000 noisy pairs of images. Each of the 50000 pairs provided corresponds to downsampled, pixelated images. The goal is to train a network that uses these two tensors to denoise i.e., reduce the effects of downsampling on unseen images. A validation set of 1000 pairs of images of the same size is also provided. The performance is assessed on Peak Signal-to-Noise Ratio (PSNR) metric. It is known that a network with 8 convolutional layers achieves 24 dB PSNR on the validation data provided and an extremely simple benchmark achieves 23 dB.

2 Implementation

The implementation of the model described in section 4 is available in the file `model.py`. In this file a class `Model` that inherits from the base class `torch.nn.Module` (see `https://pytorch.org/docs/stable/generated/torch.nn.Module.html`) is defined. The architecture of the model as well as the optimizer and the loss used during training are defined as attributes of the class during the class initialization. The forward function is explicitly defined as a class method. The `Model` provides also 3 additional methods for loading, training and predicting. More comments about implementation are available directly in the source code.

3 Building process of the model

In this section, the process of building the model is described in chronological order for clarity purpose. This section is here to motivate the choices that have been made.

3.1 Initial model

In order to validate the implementation of the `Model` class, a first simple model is implemented.

3.1.1 Network architecture

The network of the first model is composed of two convolutional layers, with an arbitrary number of channels, each followed by ReLU activation, and with two transposed convolutional layers, one followed by ReLU activation, and the other one followed by sigmoid activation function in order to range the output in value between 0 and 1, which facilitates the computation of the PSNR

metric. Weights and bias are initialized under the default method of PyTorch implementation (see https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html). Stride for each layer are set to 1 as the input image is already of a small dimension (32x32). Table 1 in appendix recaps the network architecture.

3.1.2 Loss function

The theoretical background from [1] shows that minimizing the expectation of the mean squared error (squared L2 norm) between an image with a noise ϵ and the same image with a noise δ is formally equivalent to minimizing the expectation of the mean squared error (squared L2 norm) between the image with a noise ϵ and the original clean image if ϵ and δ are different additives, unbiased, and independent noises. For this purpose the network is therefore trained under MSE loss function.

3.1.3 Training parameters

The model is trained using ADAM optimizer with a learning rate of 1e-3. The mini-batch-size is arbitrarily fixed to 50 and the model is trained over 10% of the full train set. With this configuration, this initial model already achieves 23.88 dB PSNR on the validation dataset and therefore the overall implementation is validated.

3.2 Deep Vs. Wide network

To choose the optimal number of layers for the network, multiple model with different number of convolution layers are trained over the same conditions. In order to make their performance comparable, when increasing the number of layers, the number of channels out are decrease such that every model keeps approximately the same number of trainable parameters. Table 2 in appendix recaps their structure, where the first half of the layers are convolution with kernel 3x3 followed by ReLU and the second half are transposed convolution with kernel 3x3 followed by ReLU except for the last one followed by sigmoid. The results in table 2 in appendix are most likely due to the fact that as the network gets deeper, the back propagated gradient becomes smaller and smaller in the first layer. One could try batch-normalization for deeper model but in this case, regarding PSNR after training a model with 4 layers seems to work pretty well. So to keep things simple, a model with 4 layers is kept.

3.3 Number of channels in hidden layers

In the initial model C_{out} of layers 1 and 2 are both equal. This choice has been done arbitrarily and it has been chosen to keep this structure as it was offering good results. However to increase the performance one could try to increase the number of channels. A grid search (from 10 to 400 channels with a step of 10) has been done for the number of channels in the hidden layer. Every

models has been trained on 1 epoch over 10% of the train set with a mini-batch-size of 5. Figure 1 in appendix shows the results. One can see that from $C_{out} = 100$ the increase of performance becomes less significant while the computational power to train increases drastically. Also choosing a model with fewer parameters will prevent to a possible over-fitting when training over the full train set. Therefore $C_{out} = 100$ for the hidden layers of the final model is chosen.

3.4 Skip connections

Skip connection between layer 1 and layer 4 is added to the initial model and then both models are trained over 25 epochs under the same condition than described in section 3.1.3. After training the initial model achieves 24.43 dB PSNR while the model with skip connection achieves 24.69 dB PSNR. Figure 2 in appendix shows that skip connection makes the training loss decrease faster. Thus skip connection method will be kept for the final model.

3.5 Mini-Batch-Size and Number of Epochs

A grid search is done for every mini-batch-size between 1 and 50 such that 50000 is a multiple of the mini-batch-size. The initial model is trained on 1 epoch over the full train set 5 times for every mini-batch-size and the mean PSNR is computed for each. Figure 3 in appendix shows the results. Regarding the results a mini-batch-size of 8 is kept for the final model.

Similarly a grid search between 1 and 15 with a step of 1 is done for the number of epoch for the same model with a mini-batch-size of 8 on the full train set. Figure 4 in appendix shows the results. Regarding the results, a number of training epochs of 10 is kept for the final model since beyond this point the model starts to overfit.

4 Final model and results

The architecture of the final network is shown in Table 3. MSE loss function as well ADAM optimizer with a learning rate of 1e-3 are kept. To achieve the best performance (model saved in `bestmodel.pth`) the model is trained over 10 epochs on the full dataset with a mini-batch-size of 8. The model has taken 197.95s to train on Google Colab's GPU to finally achieves of performance of 25.32 dB PSNR on the validation dataset. Results of the denoising on 3 images taken randomly in the validation dataset are shown in figure 5 in the appendix.

References

[1] Jaakko Lehtinen; Jacob Munkberg; Jon Hasselgren; Samuli Laine; Tero Karras; Miika Aittala; Timo Aila. *Noise2Noise: Learning Image Restoration without Clean Data.* `https://arxiv.org/abs/1803.04189`. 29 Oct 2018.

Appendix

Layer	C_{in}	C_{out}	Function	Activation
1	3	32	Convolution kernel 3x3	ReLU
2	32	32	Convolution kernel 3x3	ReLU
3	32	32	Transposed Convolution kernel 3x3	ReLU
4	32	3	Transposed Convolution kernel 3x3	Sigmoid

Table 1: Architecture of the initial model

Nb of layers	Channel out of each layer	Nb of parameters	PSNR after training
4	32,32,32,3	20259	23.69 dB
6	26,22,22,22,26,3	20533	22.63 dB
8	20,20,18,18,18,20,20,3	20729	20.91 dB
10	18,16,16,16,16,16,16,16,18,3	20131	20.10 dB
12	16,16,14,14,14,14,14,14,14,16,16,3	20253	19.31 dB

Table 2: Architecture with different number of layers

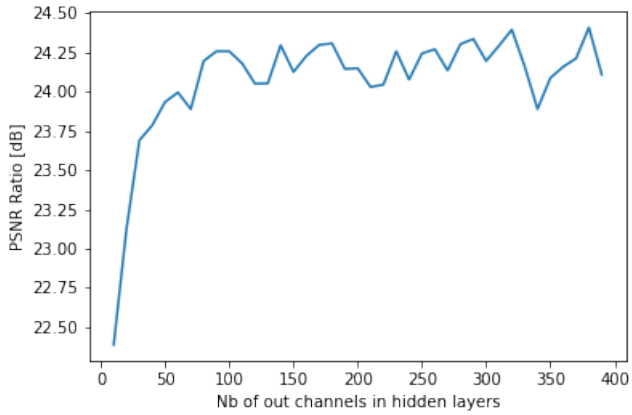

Figure 1: Performance of the model for different number of out channels in hidden layers

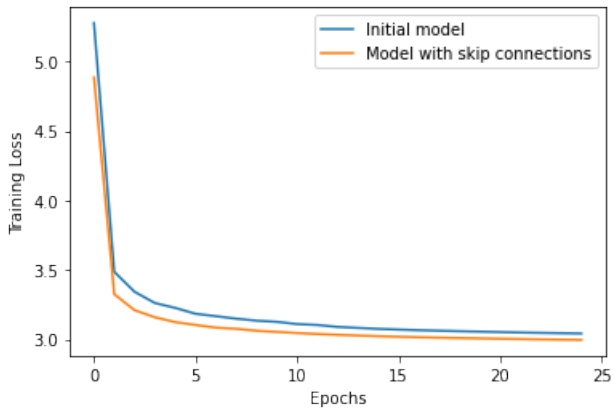

Figure 2: Comparison of model with and without skip connections

Layer	C_{in}	C_{out}	Function	Activation
1	3	100	Convolution kernel 3x3	ReLU
2	100	100	Convolution kernel 3x3	ReLU
3	100	100	Transposed Convolution kernel 3x3	ReLU
4	200 (skip connection)	3	Transposed Convolution kernel 3x3	Sigmoid

Table 3: Architecture of the final model

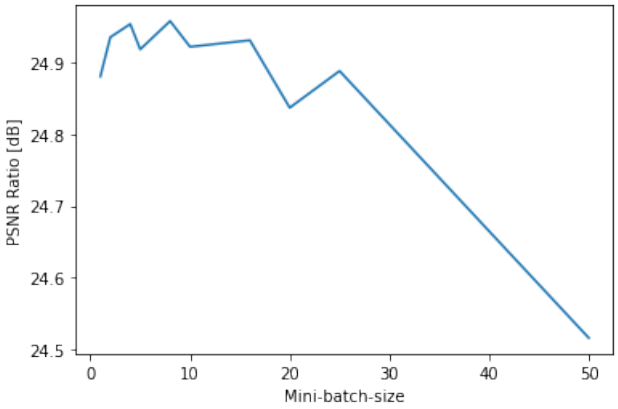

Figure 3: Mean performance of the model for different mini-batch-size

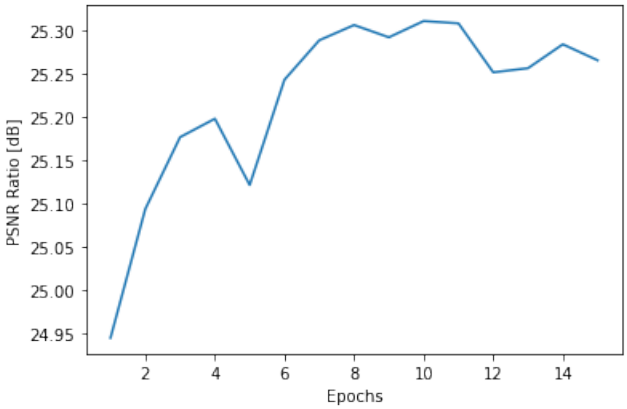

Figure 4: Mean performance of the model for different number of epoch

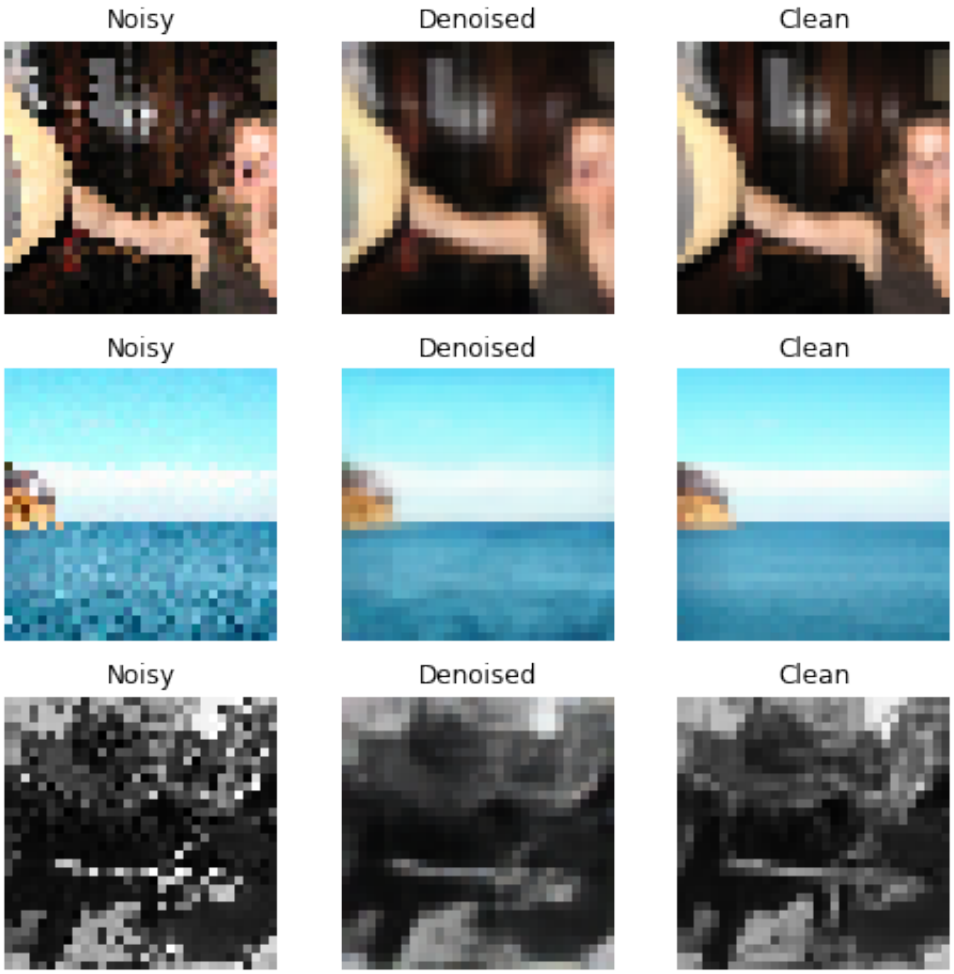

Figure 5: Results of the final model on random samples from the validation set

EE-559 Deep Learning

Mini-project 2:
Noise2Noise auto-encoder from scratch

Tom MERY **297217**

Spring 2022

Contents

List of Figures

1 Introduction

In this mini-project, a Noise2Noise model is to be implemented without using PyTorch framework. Only PyTorch tensors object and all operations that can be called directly on them as Tensor methods are allowed as well as torch.empty(), torch.cat(), torch.arange(), torch.nn.functional.fold() and torch.nn.functional.unfold().

The main purpose of this report is to explain how the whole framework has been implemented, how every block work together as well as to highlight the theoretical background when needed. No experimentation has been conducted in this mini-project, the goal is simply to implement the following network:

```
Sequential(Conv2d(stride=2),
    ReLU(),
    Conv2d(stride=2),
    ReLU(),
    Upsampling()
    ReLU(),
    Upsampling()
    Sigmoid())
```

The dataset remains the same as in the Miniproject 1.

2 Implementation

2.1 Module class

The `Module` class is the base class for every other class of the framework except for `SGD`. It is defined as follow:

```
class Module(object):
    def __init__(self):
        pass

    def forward(self, input):
        raise NotImplementedError

    def backward(self, gradwrtoutput):
        raise NotImplementedError

    def param(self):
        return []
```

Its methods will be overloaded in the child classes. For every class that inherits from Module:

- `forward()` gets for input and returns, a tensor. The input is retained in self.input when necessary.
- `backward()` gets as input a tensor or a tuple of tensors containing the gradient of the loss with respect to the module's output, accumulates the gradient with respect to the parameters, and returns a tensor containing the gradient of the loss with respect to the module's input.
- `param()` returns a list of pairs (2-list) composed of a parameter tensor and a gradient tensor of the same size. This list is empty for parameterless modules such as ReLU and Sigmoid.

2.2 Conv2d class

This class inherits from the `Module` class. The forward pass of this class is implemented as a matrix operation, using unfold (as described in the project statement). An internal function `_convolve()` (see implementation in `model.py` file) to do the convolution is therefore implemented and will be useful for both forward and backward pass as both can be seen as convolution operations. The forward pass simply does the convolution between the input of size $N \times C_{in} \times H_{in} \times W_{in}$ and the weight tensor of size $C_{out} \times C_{in} \times K_1 \times K_2$, retains the input in `self.input` and returns the output of the convolution of size $C_{out} \times C_{in} \times H_{out} \times W_{out}$ H_{out} and W_{out} are calculated as described in [1]. The weight tensor is initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where :

$$k = \frac{1}{C_{\text{in}} \times \prod_{i=0}^{1} \text{kernel\_size}[i]}$$

as in its PyTorch counterparts.

The backward pass is also implemented using convolutions:

- The gradient of the loss with respect to the input can be seen as the convolution with stride 1 of a padded, dilated version of the output gradient with a flipped version of the weight tensor (see [2]). The output gradient is padded by (`kernel_size` - 1) and dilated by (s - 1) where s is the stride used during the forward pass. The dilation and the padding can be different along the two dimensions. The weight tensor is flipped by 180 degrees on the last 2 dimensions.
- The gradient of the loss with respect to the weights can be seen as the convolution with stride 1 of the input tensor with a dilated version of the output gradient (see [2]). The output gradient is also dilated by (s - 1) where s is the stride used during the forward pass.
- The gradient of the loss with respect to the bias is simply the sum of the output gradient.

2.3 NearestUpsampling class

This class inherits from the `Module` class. The forward of this class is implemented using `repeat_interleave`. The backward is computed as the sum of the nearest element of the output gradient which actually corresponds to a convolution (with a stride = `scale_factor)` between the output gradient and a kernel of size `scale_factor`x`scale_factor` full of one. Again the convolution is implemented as a matrix multiplication.

2.4 ReLU class

This class inherits from the `Module` class. The forward pass returns:

$$ReLU(x) = \max(0, x) \quad \text{where: } x \text{ is the input} \tag{1}$$

and the backward returns:

$$\frac{dl}{dx} = \frac{dl}{do} \times \frac{do}{dx} = \frac{dl}{do} \times f(x) \quad \text{where: } f(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{if } x > 0 \end{cases} \tag{2}$$

where $\frac{dl}{do}$ is the gradient of the output given as an argument of the backward method.

2.5 Sigmoid class

This class inherits from the `Module` class. The forward pass returns:

$$o = S(x) = \frac{1}{1 + e^{-x}} \quad \text{where: } x \text{ is the input} \tag{3}$$

and the backward returns:

$$\frac{dl}{dx} = \frac{dl}{do} \times \frac{do}{dx} = \frac{dl}{do} \times \frac{e^{-x}}{(e^{-x} + 1)^2} \tag{4}$$

where $\frac{dl}{do}$ is the gradient of the output given as an argument of the backward method.

2.6 MSE class

This class inherits from the `Module` class. Compared to the other class this one takes two arguments (x,y) as input of the forward method. During the forward pass, MSE is computed as:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (x_i - y_i)^2 \tag{5}$$

and the backward pass simply returns its derivative with respect to the input:

$$\frac{d\text{MSE}}{dx} = \frac{2}{n} \sum_{i=1}^{n} (x_i - y_i) \tag{6}$$

2.7 Sequential class

This class inherits from the `Module` class. It expects a variable number of `Module` objects as arguments and store them in a list (.modules attribute).

- The forward pass consists of calling the forward methods of each module in the list with the output of the forward of the previous module.
- The backward pass consists of calling the backward methods of each module in the reversed list with the output of the backward of the previous module.

2.8 SGD class

This class implements the optimizer that uses Stochastic Gradient Descent to optimize the parameter of the `Model` class. This class has two attributes:

- `params` that stores model to be optimized.
- `lr` that stores the learning rate used.

and two methods:

- `zero_grad()` that set the gradient tensors of each parameter of the model to zero.
- `setp()` that adds the gradient tensor multiplied by the learning rate to the respective parameter tensor.

2.9 Model class

This class inherits from the `Module` class. It is implemented exactly as in Miniproject 1 except that this time the model is instantiated in the attribute .model which is a sequential object. The forward and backward pass simply call respectively the forward and backward of self.model.

3 Results

The architecture of the network is shown in Table 1. The number of channel of the network architecture is actually the same as the final model of the Miniproject 1. The learning rate has been adjusted to 0.1, the SGD optimizer is used instead of ADAM and the skip connections which are not present in this model. The network architecture is shown in Table 1 in appendix. To achieve the best performance (model saved in `bestmodel.pth`) the model is trained over 50 epochs on the full dataset with a mini-batch-size of 8. The model has taken 2760.95s to train on Google Colab's GPU to finally achieves of performance of 23.42 dB PSNR on the validation dataset. Results of the denoising on 3 images taken randomly in the validation dataset are shown in figure 1 in the appendix.

References

[1] Vincent Dumoulin; Francesco Visin. *A guide to convolution arithmetic for deep learning.* https://arxiv.org/abs/1603.07285. 23 Mar 2016.

[2] Mayank Kaushik. *Backpropagation for Convolution with Strides.* https://medium.com/@mayank.utexas/backpropagation-for-convolution-with-strides-8137e4fc2710. 3 May 2019.

Appendix

Layer	C_{in}	C_{out}	Function	Activation
1	3	100	Convolution kernel 2x2, stride 2	ReLU
2	100	100	Convolution kernel 2x2, stride 2	ReLU
3	100	100	Nearest Upsampling scale 2	
3	100	100	Convolution kernel 3x3, stride 1, padding 1	ReLU
4	100	100	Nearest Upsampling scale 2	
4	100	3	Convolution kernel 3x3, stride 1, padding 1	Sigmoid

Table 1: Architecture of the final model

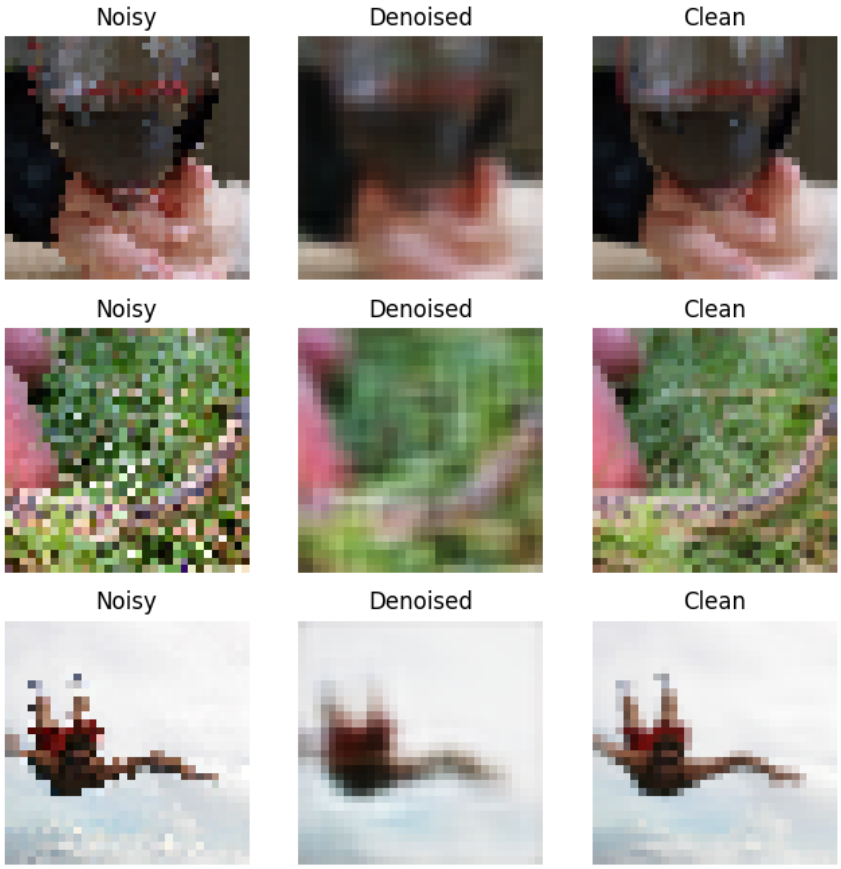

Figure 1: Results of the final model on random samples from the validation set