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

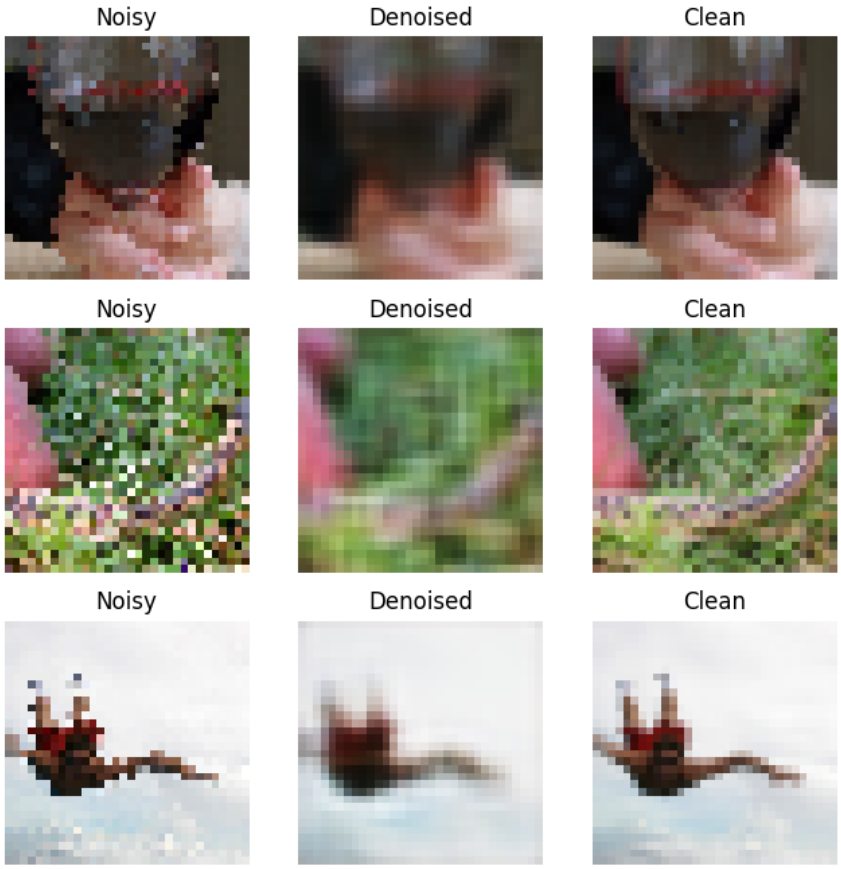

Figure 1: Results of the final model on random samples from the validation set