Mini-project 2: Write your own blocks

Arina Lozhkina, Stephen Monnet, Thibaut Stoltz
EPFL
Lausanne, Switzerland

October 31, 2022

1 Introduction

This report is part of the course *Deep Learning* given at EPFL by professor François Fleuret during the spring semester of 2022. This project aims to create Python modules to build a neural network that denoises images following the *noise2noise* method [3].

2 General Architecture

The general architecture of the neural network is that proposed by [2]. A *Model* class is used to define and train the network. It implements two convolution layers and two transposed convolution layers with several activation in between. This allows the network to get as input N $\times 3 \times 32 \times 32$ images and output N another images of the same size. This architecture is initialized using the *Sequential* class (container) that inherit from the *Module* class. The latter is used as a base class for all the modules used in the network.

3 Modules Implementation

3.1 Conv2d

The *Conv2d* module is similar to *torch.nn.Conv2d*. It can perform convolution with any batch size, 3D input, kernel size, stride and zero padding values. It has to be noted that the parameters initialization is the same as the one of *torch.nn.Conv2d*. We will detail the forward and backward pass as they were the most complicated part of the project.

Forward pass

The convolution is treated as a linear layer, as advised in the project description. To do so we first have to "unfold" the input with 'torch.nn.functional.unfold()' to get an input size of $(N, C\_in \times \prod(kernel\_size), L)$, where N is the size of the batch, C_in the number of channels of the input and L is calculated as follows in Equation 1 :

$$L = \prod_d \left[\frac{\text{spatial\_size}[d] + 2 \times \text{padding}[d] - \text{kernel\_size}[d] - 2}{\text{stride}[d]} + 1 \right] \tag{1}$$

We then reshape the weight from $(C\_out, C\_in, kernel\_size[0], kernel\_size[1])$ to a 2D tensor of size $(C\_out, C\_in \times \prod(kernel\_size))$. Now we can perform a linear convolution by doing the following matrix multiplication :

$$weight \times unfolded\_input + bias = conv\_output \tag{2}$$

It will perform the operation over the whole batch and conv_output will have the following size : (N, C_out, L). We then need to calculate the size of the output 2D image, which is found with the following formula [1]:

$$\text{out\_spatial\_size}[d] = \left[\frac{\text{in\_spatial\_size}[d] + 2 \times \text{padding}[d] - \text{kernel\_size}[d]}{\text{stride}[d]} \right] + 1 \tag{3}$$

Now that we have all the dimensions of the real output, we can finally reshape conv_output. We use 'torch.nn.functional.fold()' with a kernel size of 1×1 to do it.

Backward pass

As we treated convolution as a matrix multiplication, we simply have to implement the backward pass of a matrix multiplication. We calculate the gradients as follows in Equation 4 :

$$\text{if } A = BC, \quad \text{then} \quad \tfrac{\partial L}{\partial A} = \tfrac{\partial L}{\partial C} B^T \quad \text{and} \quad \tfrac{\partial L}{\partial B} = A^T \tfrac{\partial L}{\partial C} \tag{4}$$

With this relations, it is easy to compute the gradients of the loss w.r.t. the weight and input. The gradient of the loss w.r.t. the bias is computed by summing the gradient of the loss w.r.t. the output over its two last dimensions. Moreover, as we do batch processing, we should sum the gradients of the weight and bias over the first dimension to accumulate the gradients.

3.2 TransposeConv2d

Our *TransposeConv2d* module can then be seen as the associated transposed convolution to a prior convolution layer in the neural network. Similarly to *torch.nn.ConvTranspose2d*, it takes the same parameters (padding, kernel size and stride) as the associated convolution layer.
We choose to compute the transposed convolution as a so called *"fractionally-strided convolution"*. Indeed, we actually compute a direct convolution over a fractionally-strided input, which is the input of the transposed convolution with a number of stride -1 zeros added between each of its units. The convolution uses the same kernel_size and takes new parameter of stride and padding [1] : new_stride = 1 and new_padding$[dim]$ = kernel_size$[dim]-$padding$[dim]-1$.
The forward pass of the transposed convolution consists in adding rows and columns of zeros to the input and feeding it to the forward pass of the convolution. Similarly, the backward pass returns the reshaped output of the convolution backward pass.

3.3 Sequential

The *Sequential* class aims to group the network's modules that are given in argument. Its *forward* function calls the *forward* function from all the modules one after the other to compute the whole forward pass of the network. Its *backward* function works in the same way but it calls the backward function of each modules in the opposite order. It also manage the optimization step of each module, using the optimizer and the criterion that are also given to the class.

3.4 Loss & Activation functions

The implemented loss function l is a Mean Square Error (MSE) criterion. It takes in argument two tensors; the ground-truth tensor y_0 and the tensor to evaluate y. Its forward and backward pass simply implement equations 5, adapted for multidimensional tensor.

$$l = \frac{1}{N}\sum_{i=1}^{N}(y_0[i] - y[i])^2 \qquad \frac{\partial l}{\partial y[i]} = \frac{2}{N}(y[i] - y_0[i]) \tag{5}$$

The implemented activation functions are the Sigmoid (see 6) and the ReLU (see 7).

$$f(x) = \frac{1}{1+e^{-x}} \qquad \frac{df}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} \tag{6}$$

$$g(x) = max(0,x) \qquad \frac{dg}{dx} = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{7}$$

In our case, x is each value of the input tensor.

3.5 Stochastic Gradient Descent

The Stochastic Gradient Descent (SGD) consists of a gradient descent using only a subset of the data-set at each optimization step. In our implementation, this function is called by the *step* function of each module and receive in argument the parameters to optimize and the accumulated gradient of the loss (accumulation is done over each mini-batch) w.r.t. each of them.

4 Training of the Model

The first step of the *training* function consists of setting up the *DataLoader* which manages to create the mini-batches. Then, the training is done using two main loops to train at each epoch all the mini-batches that are part of the training set. For each mini-batch at each epoch, the following steps are done:

- Forward pass : Calling the *forward* function of the model
- Computation of the loss : Calling the *forward* function of the criterion
- Backward pass : Calling the *backward* function of the model
- Optimization step : Calling the *step* function of the model

5 Results

The architecture of the first implemented network was the following :

```
Conv2d(in=3, out=16, kernel_size=2, stride=2), ReLU(),
Conv2d(in=16, out=16, kernel_size=2, stride=2), ReLU(),
TransposeConv2d(in=16, out=16, kernel_size=2, stride=2), ReLU(),
TransposeConv2d(in=16, out=2, kernel_size=2, stride=2), Sigmoid()
```

(a) Best result obtained with the first network, PSNR = 10.52 dB

(b) Best result obtained with the second network, PSNR = 9.20 dB

Figure 1: Results obtained after 30 epochs for a learning rate $\eta = 5$

Figure 1a shows the best result obtained with this configuration. The images are disturbed by a grid of 2x2 pixels, which is probably due to our choice of stride (2). The architecture of the second network was the following :

```
Conv2d(in=3, out=16, kernel_size=2, stride=1), ReLU(),
Conv2d(in=16, out=16, kernel_size=2, stride=1), ReLU(),
TransposeConv2d(in=16, out=16, kernel_size=2, stride=1), ReLU(),
TransposeConv2d(in=16, out=3, kernel_size=2, stride=1), Sigmoid()
```

Figure 1b shows the best result obtained with a stride of 1 to avoid the grid of 2x2 pixels. Nevertheless, it only achieves a PSNR value of 9.2 dB.

References

[1] Vincent Dumoulin and Francesco Visin. “A guide to convolution arithmetic for deep learning”. In: *arXiv e-prints*, arXiv:1603.07285 (Mar. 2016), arXiv:1603.07285. arXiv: `1603.07285 [stat.ML]`.

[2] François Fleuret. *Mini-project for EE-559*. EPFL, 2022.

[3] Jaakko Lehtinen et al. “Noise2Noise: Learning image restoration without clean data”. In: *arXiv preprint arXiv:1803.04189* (2018).