Mini-project 2: Image Denoising Network

Maximilian Gangloff, Gabriel Maquignaz, Mihaela Berezantev
EE-559 Deep Learning, EPFL, Switzerland

***Abstract*—After exploiting the PyTorch framework to build our network for denoising images, we wrote our own blocks without using autograd or torch.nn modules. This network, although considerably simplified, compared to the one from mini-project 1, also learns to remove noise from images after applying a series of convolutions and activation functions on noisy data. The aim of this paper is to show the complete procedure needed to obtain the final model by presenting chosen implementation and architecture approaches.**

I. Introduction

Image denoising is the process of removing noise from a noisy image, in order to restore the true image. Various techniques including statistical modeling of signal corruptions, generative learning based methods, etc. aim at achieving this goal, but this task is still a challenging and open problem. This project focuses on using a convolutional neural network that learns to map corrupted inputs to clean images by minimizing a loss function between these two. The quality of the results can be measured using the Peak signal-to-noise ratio (PSNR) - an expression for the ratio between the maximum possible value of a signal and the power of distorting noise that affects the quality of its representation [1].

II. Modules

In order to construct the model and be able to train it on the dataset with noisy images and corresponding clean targets, we need to implement the following modules using the Python Standard Library: Mean Squared Error(MSE), Sigmoid, ReLU, Convolution, Upsampling, Stochastic Gradient Descent, a container similar to `torch`'s `Sequential`. Finally, we will need to put them all together so as to construct the final model to be trained. Apart from the Python Standard Library, additional tensor operations are allowed, more precisely all `.foo()` methods that can be applied on tensors. Also, `fold` and `unfold` from `torch.nn.functional` are used for implementing the convolutions.

The chosen architecture was to define a class `Module` that implements a `forward`, `backward`, and `param` function and let all the modules inherit from it. In the following sections, our choices for each method of these modules are explained in more details.

A. Mean Squared Error

The `forward` of this loss function computes the average squared difference between the estimated value $\hat{y}$ and the actual value y, i.e. $\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y)^2$. As for the `backward` pass, it computes the derivative of the loss with respect to a predicted value $\hat{y}$, i.e., $\frac{2}{n}(\hat{y}_i - y)$.

B. Sigmoid

The `Sigmoid` (σ) is used as an activation function and its `forward` computes for every input x (here we assume it's a scalar value for simplicity) the value $\frac{1}{1+e^{-x}}$. As for the `backward`, it returns the gradient of the loss with respect to the input by making use of the chain rule and the gradient of the loss with respect to the output y. Assuming scalar values, we have $\frac{\partial l}{\partial x} = \frac{\partial l}{\partial y}\frac{\partial y}{\partial x} = \frac{\partial l}{\partial y}\sigma(x)(1-\sigma(x))$.

C. ReLU

The `ReLU` is another activation function defined as the positive part of the input - `max(0, x)` - which corresponds to the `forward` method. The `backward` is based on gradient computation similar to that of the `Sigmoid` and returns $\frac{\partial l}{\partial y}$ if $x > 0$ and 0 otherwise because the derivative of `ReLU` is 1 when $x > 0$ and 0 otherwise.

D. Convolution

Mathematically, a convolution can be seen as a linear function, i.e. the multiplication of the reshaped input by a weight matrix, plus a bias element. These two are the parameters of the convolution that are initialized with random values from the normal distribution with mean 0 and a standard deviation of 1. In the `forward` method we use `unfold` to reshape the input and return y after doing the convolution. The output needs to have a specific shape that depends on the input's size, the given padding, dilation, stride, and kernel (the formulas used for computing the output shape can be found here [2]). We obtain y by doing $y = weight \cdot unfolded\_x + bias$.

In the `backward` pass, we need to update the gradient of the loss with respect to the parameters (weight, bias) as well as return the gradient of the loss with respect to the input x, using the input of the function - the gradient of the loss with respect to the output. We use the following formulas:

- $\frac{\partial l}{\partial bias} = \frac{\partial l}{\partial y} \cdot \frac{\partial y}{\partial bias} = \sum_{i=1}^{n} \frac{\partial l}{\partial y_i}\frac{\partial y_i}{\partial bias} = \sum_{i=1}^{n} \frac{\partial l}{\partial y_i}$
- $\frac{\partial l}{\partial weight} = \frac{\partial l}{\partial y} \cdot \frac{\partial y}{\partial weight} = \frac{\partial l}{\partial y} \cdot unfolded\_x$
- $\frac{\partial l}{\partial unfolded\_x} = \frac{\partial l}{\partial y} \cdot \frac{\partial y}{\partial unfolded\_x} = \frac{\partial l}{\partial y} \cdot weight$

Finally, we fold $\frac{\partial l}{\partial unfolded\_x}$ in order to obtain the needed result $\frac{\partial l}{\partial x}$.

[1] https://www.ni.com/fr-fr/innovations/white-papers/11/peak-signal-to-noise-ratio-as-an-image-quality-metric.html

[2] https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html

E. Upsampling

Upsampling in a neural network is usually done with transpose convolutions. However, we chose to implement the upsampling module using a Nearest-neighbor upsampling (NNU) followed by a convolution as while not equivalent mathematically, it is shown to fulfill the same role within a network.

For the forward pass, the input signal is first upsampled and then passed to a regular 2D convolution as described in Section II-D. For the backward pass, because the NNU has no trainable parameters, the gradient with respect to the parameters is exactly that returned by the 2D convolution. The gradient with respect to the input, however, must be downsampled to have the correct shape and be used by the rest of the network.

Nearest-neighbor upsampling can be implemented in a number of different ways.

$$\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \longrightarrow \begin{bmatrix} m_{11} & m_{11} & m_{12} & m_{12} \\ m_{11} & m_{11} & m_{12} & m_{12} \\ m_{21} & m_{21} & m_{22} & m_{22} \\ m_{21} & m_{21} & m_{22} & m_{22} \end{bmatrix}$$

Fig. 1. Nearest-neighbor upsampling with a scale factor of 2.

For the sake of efficiency and elegance we implemented the NNU as two matrix multiplications. One can, indeed easily prove that any NNU can be applied to an input matrix $\boldsymbol{X}$ by multiplying it left and right by two specific matrices. Let $\boldsymbol{X}_{up}$ denote the upsampled version of the input $\boldsymbol{X}$ (i.e. the result of the upsampling). Then $\boldsymbol{U}_l$ and $\boldsymbol{U}_r$ are respectively the left and right upsampling matrices so that

$$\boldsymbol{X}_{up} = \boldsymbol{U}_l \cdot \boldsymbol{X} \cdot \boldsymbol{U}_r \tag{1}$$

Let $\alpha \in \mathbb{N}^*$ be the scaling factor and let $\boldsymbol{\mathcal{I}}_{N,\alpha}$ denote the identity matrix of initial size $N \times N$ dilated by α along the first dimension (i.e. where every element has been replaced by α copies of themselves arranged along the first dimension). An example is given below.

$$\boldsymbol{I}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \boldsymbol{\mathcal{I}}_{3,2} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

Fig. 2. Normal 3x3 identity matrix (left) and dilated identity matrix with α = 2 (right).

Finally, let us consider an input $\boldsymbol{X}$ of size $M \times N$ and an arbitrary scaling factor α. The left and right upsampling matrices are then given by the following.

$$\boldsymbol{U}_l = \boldsymbol{\mathcal{I}}_{M,\alpha}$$
$$\boldsymbol{U}_r = \boldsymbol{\mathcal{I}}_{N,\alpha}^T$$

In addition, even though the upsampling matrices are not square and thus cannot be inverted, the upsampling can be easily reverted by noticing the following.

$$\boldsymbol{\mathcal{I}}_{N,\alpha}^T \cdot \boldsymbol{\mathcal{I}}_{N,\alpha} = \alpha \cdot \boldsymbol{I}_N, \quad \forall N, \alpha \in \mathbb{N}^* \tag{2}$$

It follows that without inverting any matrices, the upsampling can be reverted like so.

$$\boldsymbol{X}_{up} = \boldsymbol{U}_l \cdot \boldsymbol{X} \cdot \boldsymbol{U}_r \iff \boldsymbol{X} = \frac{1}{\alpha^2} \left(\boldsymbol{U}_l^T \cdot \boldsymbol{X}_{up} \cdot \boldsymbol{U}_r^T \right) \tag{3}$$

In the case of the gradient with respect to the input, this will be equivalent to averaging adjacent gradients to reshape the gradient tensor which is reasonable.

To avoid any `for` loops, the upsampling matrices are computed by filling a 1D tensor with zeros, putting ones at the correct indices and then reshaping it correctly using `.view()`. The set $\mathbb{S}$ of indices of the 1D tensor containing a one is easily computed as follows.

$$\mathbb{S} = \{ i \in [0, \alpha D^2[\mid i \bmod \alpha(D+1) < \alpha \} \tag{4}$$

with D the size of the non-dilated identity matrix (i.e. for $\boldsymbol{U}_l$: $D = M$ and for $\boldsymbol{U}_r$: $D = N$).

F. Stochastic Gradient Descent

We implement Stochastic Gradient Descent (SGD) that will be used to optimize the model parameters. It does not inherit from Module as it only needs a `step()` function that will update all the model parameters using a learning rate (α) chosen by the user. In the case of the weight, for example, the update is $w := w - \alpha \nabla L(w)$. Also, a function `update_parameters()` is necessary in order to be able to update the gradients that the SGD uses in the `step` function after they are updated in the `backward()` of the convolution or the upsampling.

G. Container

This module is equivalent to PyTorch's `Sequential` and is used for putting together an arbitrary configuration of modules together. The `forward()` method recursively calls the `fowrard()` of each module. The `backward()` function is similar, except that the order of the models is reversed as we need to pass the gradients upstream.

H. Model

Finally, everything is put together in the `Model` class. The built network contains the following blocks in this specific order:

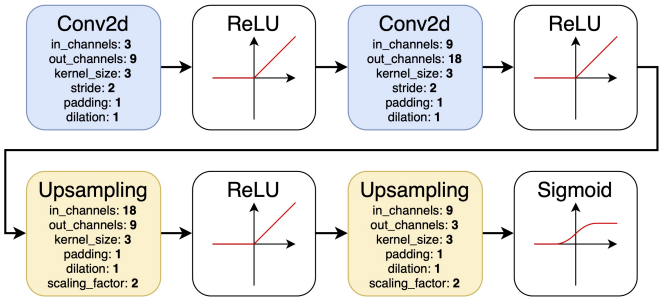

Fig. 3. Network architecture.

III. Results

Each module has been tested individually using its PyTorch equivalent and is working as intended. When putting everything together and training the model, we have experienced problems with vanishing gradient and could not train our model. We have tested everything to solve this problem including changing the distribution of the initialisation of the parameters, tuning the learning rate over several orders of magnitude, tuning the batch size and replacing the last sigmoid module by a ReLU module. We could not get it to work. This surely underlines the sensitivity to parameters initialisation and hyper-parameters tuning in this field.

IV. Conclusion

This mini-project allowed us to get a true sense of what is found under the hood when using PyTorch and Autograd. Even though we did not manage to achieve a denoising, we still feel like this really helped us get a better grasp of the implementation challenges and the complexity of both designing the architecture and tuning the parameters of such networks.