# Miniproject 2

Luca Gravina Giorgio Palermo

## Introduction

Through basic statistical arguments it has been shown, and discussed in Report 1, that signal restoration by machine learning can be performed even with a data-set consisting exclusively of corrupted samples; the only requirement being that each pair of samples in the data-set (training input and training target) explicates independent and identically distributed realizations of some noise model.

While the focus of Report 1 was set on the realization of a neural network (NN) reconstructing an image with the highest possible peak signal-to-noise ratio (PSNR) evaluated on a validation set composed of a noisy image (input) and its ground truth (target), here we focus on the custom implementation of each of the code blocks used to realize the above.

The simple sequential network shown in Figure (1), composed of two convolutional layers and two transposed convolution upsampling layers, is used to benchmark our implementation of the aforementioned functions. Rectified linear units (ReLU) are employed as nonlinearities between the inner layers, while a final Sigmoid function is employed, as in Report 1, to bound the output of the network within the interval $[0, 1]$.

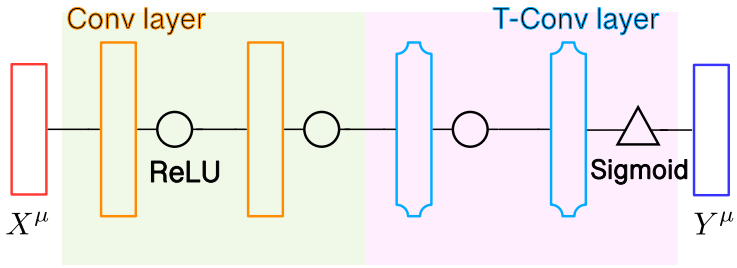


Figure 1: Simple sequential NN for noise-to-noise signal reconstruction.

## (Transposed) Convolution layer

A set of $N$ digital images $\{X^\mu\}_{\mu\in\{1,\dots N\}}$ of height $H$ and width $W$ can be represented as a tensor $(X^\mu)^\alpha_{mn}$ with the indices $m \in [0, H-1], n \in [0, W-1]$ indicating pixel coordinates, $\alpha \in \{\mathrm{R}, \mathrm{G}, \mathrm{B}\}$ the color channel, and $\mu$ the selected sample. The convolution operation of this tensor with the four-dimensional kernel $f^{ij}_{\alpha\beta}$ of size $h \times w$, $D$ features and $C$ input channels, is defined as:

$$(Y^\mu)^\beta_{mn} = (X^\mu \circledast f)^\beta_{mn} = \sum_\alpha \sum_{i,j} f^{\alpha\beta}_{ij}\ (X^\mu)^\alpha_{m+i\,n+j} \quad \text{with} \quad \begin{cases} i \in [0, h-1], \\ j \in [0, w-1], \\ \beta \in [0, D-1] \end{cases} . \tag{1}$$

Given a loss function $\mathcal{L}$ evaluated on the output of a convolution layer $(Y^\mu)^\alpha_{kl}$, its derivative with respect to the convolution's input $(X^\mu)^\alpha_{mn}$ and weights $f^{\alpha\beta}_{ij}$ is found to be:

$$\frac{\partial \mathcal{L}}{\partial\,(X^\mu)^\alpha_{mn}} = \sum_\beta \sum_{k,l} \frac{\partial \mathcal{L}}{\partial\,(Y^\mu)^\beta_{kl}} f^{\alpha\beta}_{m-k\ n-l} = \left(\frac{\partial \mathcal{L}}{\partial Y^\mu} * f\right)^\alpha_{mn}, \tag{2}$$

$$\frac{\partial \mathcal{L}}{\partial f^{\alpha\beta}_{ij}} = \sum_\mu \sum_{i,j} \frac{\partial \mathcal{L}}{\partial\,(Y^\mu)^\beta_{ij}}\ (X^\mu)^\alpha_{m+i\ n+j} = \sum_\mu \sum_{i,j} \frac{\partial \mathcal{L}}{\partial (\tilde{Y}^\beta)^\mu_{ij}}\ (\tilde{X}^\alpha)^\mu_{m+i\ n+j} = \left(\frac{\partial \mathcal{L}}{\partial \tilde{Y}^\beta} \circledast \tilde{X}\right)^\alpha_{ij}. \tag{3}$$

Here $\tilde{Y}$ and $\tilde{X}$ identify a transposition operation along the sample and channel dimensions of the four-dimensional tensors $(Y^\mu)^\beta_{ij}$ and $(X^\mu)^\alpha_{mn}$. The operation defined in (2) is known as a *transposed convolution* and can be written for a general input tensor $(X^\mu)^\alpha_{mn}$ and kernel $f^{\alpha\beta}_{ij}$ as:

$$(Y^\mu)^\beta_{mn} = (X^\mu * f)^\beta_{mn} = \sum_\alpha \sum_{i,j} f^{\alpha\beta}_{ij}\ (X^\mu)^\alpha_{m-i\,n-j}\,. \tag{4}$$

Having shown that all derivatives relevant to a (transposed) convolutional layer can be cast as convolutions and transposed convolutions themselves, two main points remain to be examined, namely: an efficient implementation of a (transposed) convolutional layer with arbitrary stride, padding, kernel size and dilation, and the effect of a non-unit stride on Eqs. (2) and (3).

The former, consists in reducing the problem to a single matrix multiplication via `PyTorch`'s `fold` and `unfold` methods. Consider the convolution $Y^\mu = X^\mu \circledast f$ with kernel $f \sim [D, C, h, h]$. Within each sample image $X^\mu \sim [1, C, H, H]$, we identify all those $h \times h$ patches which contribute to the convolution's output via the Frobenius inner product, convert them into column arrays, and stack them row-wise. The presence of several input channels is accounted for by column-stacking patches overlapping along the channel dimension. The resulting flattened image is $X^\mu \rightsquigarrow X'_\mu \sim [1, h^2\, C, L]$, with

$$L = \left( \left\lfloor \frac{H - 2 \cdot \text{padding} - \text{dilation} \cdot (\text{kernel}_\text{s}\text{ize} - 1) - 1}{\text{stride}} + 1 \right\rfloor \right)^2 . \tag{5}$$

The presence of multiple samples is accounted for by concatenating the first and last dimension of $X'$. Finally, $(X')^\mu \sim [NL, h^2\, C]$. At the same time, the kernel tensor is flattened along the last three dimensions so that $f \rightsquigarrow f' \sim [D, h^2\, C]$. The convolution operation can now be written as a simple matrix multiplication: $Y'_\mu = (X')^\mu \cdot (f')^T$ with $(Y')^\mu \sim [NL, D]$. Lastly, the output vector can be reshaped into the four dimensional tensor $Y : [N, D, \sqrt{L}, \sqrt{L}]$. Similarly, any transposed convolution operation $Z = Y^\mu * f$ can be reduced to the matrix multiplication $Z' = (Y')^\mu \cdot f'$ where $Z'$ is the flattened representation of the output $Z$ whose reshaping operations are carried out in reversed order with respect to the previous case.

Finally, non-unit strides can be easily integrated into the backward propagation by simply dilating the matrix $\partial L / \partial Y^\mu$ by $s = \text{stride} - 1$. With a non-unit stride, the possibility of not covering the input map entirely arises. In such cases, the derivatives with respect to some of the input values have to be manually set to vanish.

## Module class

Each of the four different types of layers (Convolution, Transposed Convolution, Sigmoid and ReLU) share the same structure implementing the forward and backward pass.

The weights and biases of each module, should there be any, are stored as instance variables in `module.weight` and `module.bias`, respectively, and are updated by the stochastic gradient descent (SGD) algorithm. The latter requires the derivative of a user-defined loss function with respect to the layer's weights and biases, which are evaluated at each step and stored into `module.d_weight` and `module.d_bias`.

Two methods are common to all modules: `forward` and `forward_and_vjp`. The forward method takes as input a tensor, to which it applies a different transformation depending on the type of layer (see below), and returns the transformed tensor.

The `forward_and_vjp` method takes the same arguments as the `forward` method and returns a tuple, the first element of which is identical to the output of the `forward` method alone. The second element of this tuple, on the other hand, consists of the function handle `_vjp(torch.Tensor)` which, given the derivative of the loss function with respect to the output of the module, computes the derivative of the loss with respect to the module's input, parameters and bias, namely $\partial \mathcal{L} / \partial X^{(\ell)}$ and $\partial \mathcal{L} / \partial f^{(\ell)}$ and $\partial \mathcal{L} / \partial b^{(\ell)}$, where $\ell$ identifies a specific layer. As we have seen, although $\partial \mathcal{L} / \partial X^{(\ell)}$ can be computed through $\partial \mathcal{L} / \partial Y^{(\ell)}$ alone, computing $\partial \mathcal{L} / \partial f^{(\ell)}$ requires the knowledge of the input $X^{(\ell)}$ of each layer. Therefore, when moving though the forward pass, the latter would have to be stored in an additional layer-specific instance variable along with any other quantity necessary for the evaluation of the layer's backward pass. The advantage of returning the function `_vjp` rather than the layer's derivatives directly, is that all necessary quantities for its evaluation (which all possess a higher or equal scope than that of the function's definition) are automatically stored within the function handle. The implementation of the Sigmoid layer is shown below as a simple example of the aforementioned structure.

```
class Sigmoid(Module):
    ...
    def forward(self,input):
        return torch.sigmoid(input)
    __call__ = forward

    def forward_and_vjp(self, input):
        def _vjp(dL_dy):
            dsigma_dx = torch.sigmoid(input)*(1.-torch.sigmoid(input))
            return (dL_dy*dsigma_dx , torch.Tensor([]), torch.Tensor([]))
        return self.forward(input), _vjp
```

## Container

The `Sequential` class implements the analogous of `PyTorch`'s sequential container. Modules are added to it, and stored within the ordered dictionary `self._modules`, in the order they are passed in the constructor. The whole container is treated as a single module, and as such, it presents with both a `forward` and a `forward_and_vjp` method.

The `forward` method accepts as input a four-dimensional tensor which is passed directly to the first module of the container. The output of each module is taken as the input of the next one, finally returning the output of the last module (see below).

```
def forward(self, input):
    for module in self._modules.values():
        input = module(input)
    return input
```

In much the same way, the `forward_and_vjp` method sequentially accesses each module in the container, evaluating both its output and the associated function handle `_vjp`. While the former is again used as input to the subsequent layer, the latter is stored within a list of functions which, upon reversed traversal, is able to produce all derivatives necessary for the backward pass. The following code snippet is explicative of the aforementioned process.

```
def forward_and_vjp(self, input, vjp_loss):
    VJP = [None]*self.nb_modules
    for i,module in enumerate(self._modules.values()):
        input, VJP[i] = module.forward_and_vjp(input)

    dL_dy = vjp_loss(input)
    for i, (module, vjp) in enumerate(zip( reversed(self._modules.values()), reversed(VJP) )):
        dL_dy, module.d_weight, module.d_bias = vjp(dL_dy)
```

Noticeably, the derivative of the loss with respect to a layer's input $\partial\mathcal{L}/\partial X^{(\ell+1)}$, is used as the previous layer's derivative with respect to its output $\partial\mathcal{L}/\partial Y^{(\ell)}$, fundamental for the propagation of the backward pass. Initiating this process of concatenated function evaluations, is the evaluation of the loss' derivative with respect to the network's output. In our code we implemented the MSE loss, $\mathcal{L} \propto \sum_\mu \|X^\mu - Y^\mu\|_2$ so that $\partial\mathcal{L}/\partial Y^\mu \propto 2(X^\mu - Y^\mu)$. Both the output of the forward pass and of the backward pass of the convolutional layers were compared to the ones obtained from `PyTorch`'s native implementation using the `torch.allclose` function. In particular, each of the derivatives were tested separately by comparing the output of `_vjp` to the elements of `torch.grad(F.mse_loss(y, F.conv2d(x,weight=f, bias=b)), [x,f,b])`.

## Network results

To summarize our results we display in Figure (2) the PSNR for both the custom and `torch.nn` implementations of the NN. Upon sufficient training ($> 4$ epochs), we find $\text{PSNR}_{\text{custom}} = 23.55 \pm 0.18$ and $\text{PSNR}_{\text{torch.nn}} = 23.68 \pm 0.11$.

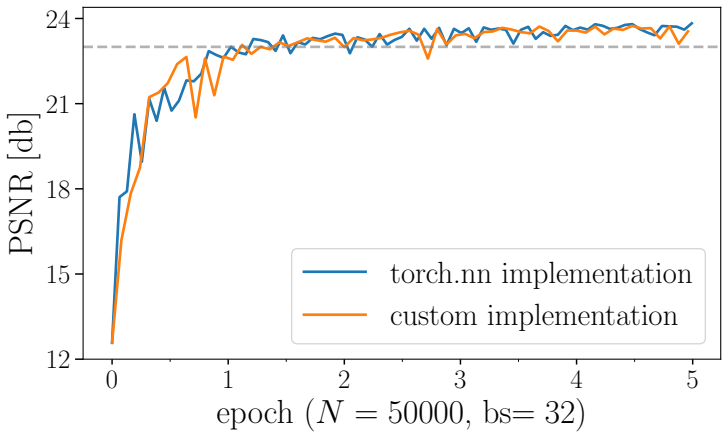


Figure 2: PSNR ratio for both the custom and standard implementations of the NN displayed in Figure (1).