# Miniproject 1

Luca Gravina Giorgio Palermo

## Introduction

Signal restoration and statistical modelling of signal corruptions, have been long-lasting problems in statistical data analysis, to which, recently, deep neural networks have been successfully applied. Typically, a regression model, e.g., a convolutional neural network (CNN), is trained on a large data-set consisting of pairs $(X^\mu, Y^\mu_{\text{clean}})$ of corrupted inputs $X^\mu$ and clean targets $Y^\mu_{\text{clean}}$. By minimizing an appropriately chosen empirical risk function $\mathcal{L}$, a parametric family of mappings between a noisy input and the underlying ground truth to be inferred, is so found.

In this work we follow the observations put forward by Lehtinen et al. (2018), and remark that signal reconstruction, and training of a regression model, can be performed even on a data-set consisting exclusively of corrupted samples, with comparable - if not superior - results than those obtained using clean references.

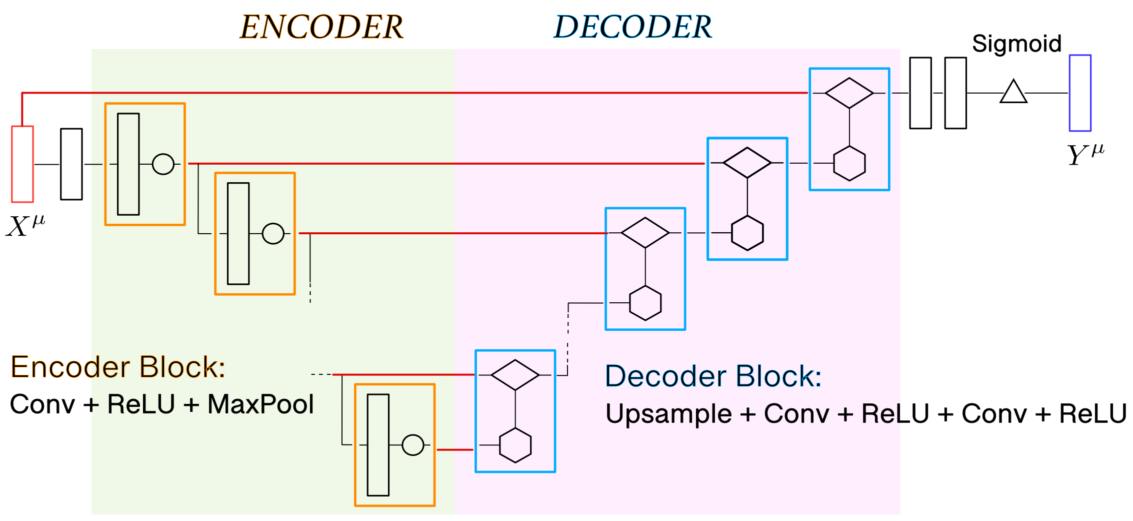

Figure 1: Network architecture used in our report. We opted for `PReLU` nonlinearities rather than the more common `LeakyReLU` to increase the number of learnable parameters without having to increase the number of convolutional and pooling layers as well.

## Network description

The chosen autoencoder architecture performing signal restoration is displayed in Figure (1) and was designed to operate on input tensors with dimension $X^\mu \sim [N, C = 3, H = 32, W = 32]$.

A first convolutional layer separates the input from the encoding structure (green shaded area), and serves the purpose of increasing the number of relevant features from $C = 3$ to $C_e$. The encoding structure is composed of $N_e = 4$ encoding blocks (orange rectangles), each one consisting of a convolution operation with kernel $f_e \sim [C_e, C_e, h = 3, w = 3]$, a Max-Pooling operation with kernel $f_{mp} \sim [2, 2]$, separated by a ReLU nonlinearity with trainable parameters (`PReLU`). All convolution operations are accompanied by an appropriate padding, so as to keep the image's spatial dimensions unchanged, and avoid boundary-induced artifacts in the image reconstruction. The unavoidable compression of the latent space, however, is what constrains the number of possible encoding layers to $N_e$.

$N_e$ decoder blocks (blue rectangles) are used to create the decoding structure shown as a pink shaded area in Figure (1). Each block consists of an upsampling layer with scaling factor $s_{\text{up}} = 2$, a concatenation layer, and two convolutional layers with kernels $f_{d,1} \sim [C_{d,1}, C_{d,1}, h = 3, w = 3]$ and $f_{o,2} \sim [C_{d,2}, C_{d,3}, h = 3, w = 3]$ respectively. The latter alternate with as many trainable `ReLU` nonlinearities. The following code details the implementation of a generic decoding block. Note that not all blocks will share the same number of features, as this depends on the number of channels carried by the second input of the concatenation layer.

```
class _Decoder_Block(nn.Module):
    def __init__(self, in0, in1, out1, kernel_size):
        super().__init__()
        self.conv0 = nn.Conv2d(in0, in1 , kernel_size padding='same')
        self.conv1 = nn.Conv2d(in1, out1, kernel_size, padding='same')
        self.relu0 = nn.PReLU(in1)
        self.relu1 = nn.PReLU(out1)

    def forward(self, x, y):
        x = F.interpolate(x, scale_factor=2, mode='nearest') #upsample
        x = torch.cat((x,y),dim=1)     #concatenate
        x = self.relu0(self.conv0(x)) #first convolution
        x = self.relu1(self.conv1(x)) #second convlution
        return x
```

Progressing through the forward pass of the network, the original input, and the output of all encoder blocks but the last, are stored in memory (red lines). The concatenation layers of the decoding blocks are progressively used to concatenate the latter to the advancing upsampled signal, taking care to match those tensors with analogous dimensions. Two convolutional layers are placed at the output of the decoding structure to restore the number of features to $C = 3$; while a final Sigmoid function is employed to bound the output of the network between 0 and 1. The inner workings of the network are best understood by looking at its `PyTorch` implementation, which is hereunder displayed.

```
class Model(nn.Module):
    def __init__(self):
        super().__init__()
        ...
        #ENCODER
        self.conv0 = nn.Conv2d(ChIm, oute, kernel_size, padding='same')
        self.conv1 = nn.Conv2d(oute, oute, kernel_size, padding='same')
        eblock = _Encoder_Block(oute, oute, kernel_size, maxp_ksize=2)
        self.eblocks = nn.ModuleList([eblock]*Ne)

        #DECODER
        dblock0 = _Decoder_Block(in0=2*oute, in1=outd, out1=outd, kernel_size)
        dblock1 = _Decoder_Block(in0=outd+oute, in1=outd, out1=outd, kernel_size)
        dblock2 = _Decoder_Block(in0=outd+3, in1=outd//2, out1=outd//3, kernel_size)
        self.dblocks = nn.ModuleList([dblock0] + [dblock1]*(nb_elayers-2) + [dblock2])

        self.conv2 = nn.Conv2d(outd//3, 3, kernel_size=kers, padding='same')
        self.relu  = nn.PReLU()

    def predict(self, x):
        ...
        #ENCODER
        pout = [x]
        y = self.relu(self.conv0(x))
        for l in self.eblocks[:-1]:
            y = l(y)
            pout.append(y)
        y = self.eblocks[-1](y)
        y = self.relu(self.conv1(y))

        #DECODER
        for i,l in enumerate(self.dblocks):
            y = l(y, pout[-(i+1)])
        y = torch.sigmoid(self.conv2(y))
        return y
```

## Results

The figure of merit chosen to benchmark our network is the peak signal-to-noise ratio (PSNR) which is evaluated on a validation set composed of $N_{\text{val}} = 1000$ pairs of noisy images (to be denoised by the network) and their associated ground truths.

The model's parameters are trained by the stochastic gradient descent optimizer for which a vanishing weight decay has been chosen, as suggested when in the presence of `PReLU` nonlinearities. The input is decomposed in batches of size $\text{bs} = 16$ and the validation of the model is performed every $n_{\text{bs}} = 150$ batches. The MSE loss generated by the validation procedure is fed into `PyTorch`'s `torch.optim.lr_scheduler.ReduceLROnPlateau()`

scheduler so as to dynamically reduce the learning rate in response to changes in the PSNR and delay as much as possible its saturation. In order to make the input variables be as decorrelated as possible, we compute mean and standard deviation "per pixel and color channel" and normalize the data with respect to it. We observed that the benefit introduced by such pre-processing of the data is analogous to that provided by a "per image" normalization.

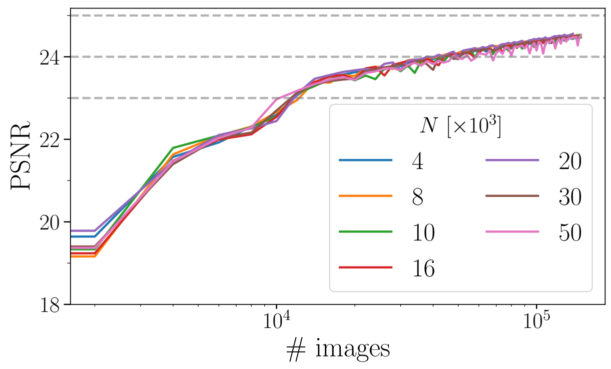

Figure 2: PSNR evolution as a function of the number of processed images. The choice of using the number of images in the training set (instead of the more common number of epochs) as coordinate for the horizontal axis is motivated by the desire to compare the network's denoising capabilities when trained with datasets of different sizes.

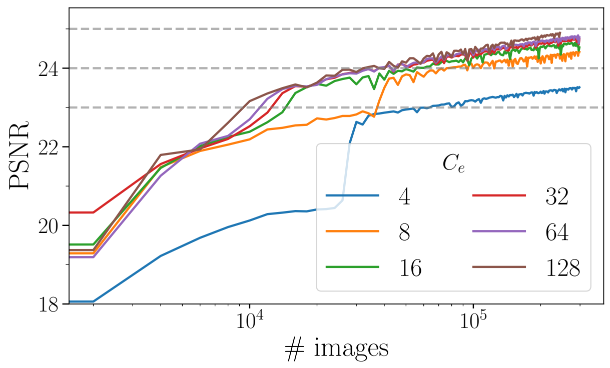

Figure 3: Dependence of the PSNR's evolution on the number of features $C_e$ embedded within each encoding layer. In a regular scale, the above curves are analogous to those found by Lehtinen et al. (2018) apart from a lower PSNR which is most likely due to the reduced native image resolution ($32\times32$ vs $256\times256$) and (consequently) shallower network.

In Figure (2) we display the behaviour of the PSNR against the number of processed (possibly repeating, e.g, when epoch $> 1$) images for different truncations of the training dataset. We observe that increasing the number of independent images beyond $\sim 4 \times 10^3$ does not result in an increase in the PSNR: indeed, although more epochs are required when considering smaller datasets, identical precisions are reached upon processing a fixed, sufficiently large, number of images. For completeness, we performed data augmentation (rotations, color jittering and cropping) on the training datatset and evaluated the evolution of the PSNR over $N = 150000$ independent samples. Curves identical to those presented above were found.

Noticeably, both the model's learning rate and its ultimate precision are influenced by the number of features $C_e$ embedded within each encoding block. In the native network we set $C_e = 32$ as models with $\gtrsim 32$ features seem to perform almost identically, [c.f. Figure (3)].

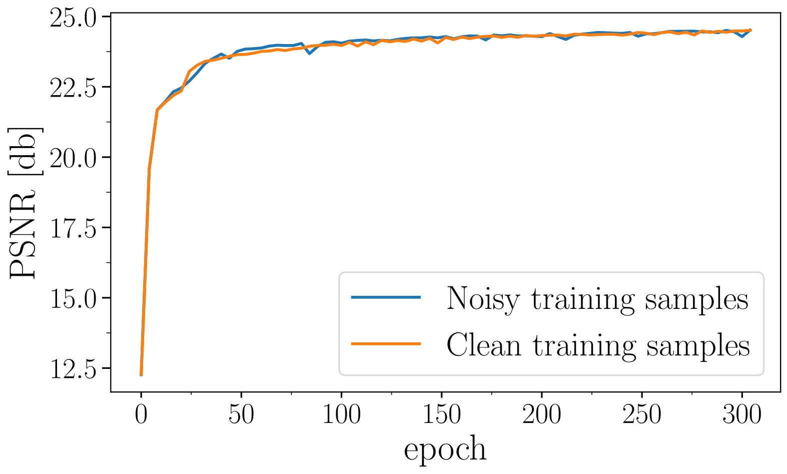

Figure 4: Comparison of the PSNR's increase during training for clean and corrupted target references. As expected, no advantage is observed when training over clean references.

We remark that values of PSNR $\sim 25$ can be reached with the network in Figure (1). This is to be compared to the simpler network implemented in Report 2 whose efficiency is limited to PSNR $\sim 23$.

To conclude, we empirically verify our initial statement, that is, that a denoising network trained on a dataset composed exclusively of noisy samples is as performing as one trained over a dataset consisting of pairs of corrupted inputs, and clean targets. To do so, we split in half the validation set: the first half we use to train the model, the second for validation. The same model is then trained on 500 pairs of compromised images. Surely, in Figure (4) we are able to observe the equivalence in performance.

### Further observations

On a quest to improve the network proposed in Figure (1) we considered networks containing batch normalization and dropout layers in different locations, and tested their effectiveness under different choices of parameters. No improvement was registered, we suppose, because of an insufficient depth of the network.

Although the weight's evolution rates across layers during training were not directly observed, we verified that no remarkable improvement is provided by a weight initialization different from `PyTorch`'s native one.

## References

Jaakko Lehtinen, Jacob Munkberg, Jon Hasselgren, Samuli Laine, Tero Karras, Miika Aittala, and Timo Aila. Noise2noise: Learning image restoration without clean data, 2018. URL `https://arxiv.org/abs/1803.04189`.