TP1

- Master MVA ENS-Paris Saclay
- Balthazar Neveu
- balthazarneveu@gmail.com

DCT denoiser

Question 1.

Maximizing the likelihood

Given $Y = X + B$ where $B \sim \mathcal{N}(0, \sigma^2)$ and $p(X) = \frac{e^{-\|X\|}}{\int_{-\infty}^{\infty} e^{-\|X\|} dx}$

We're looking for the most likely value x of X given the observation y of the random variable Y.

$x^* = \text{argmax}_x P(X = x | Y = y)$

Let's first apply Bayes rule

$$P(X = x|Y = y) = \frac{P(Y=y|X=x)P(Y=y)}{P(X=x)} = \frac{P(B=y-x|X=x)P(Y=y)}{P(X=x)} \propto \frac{e^{-\frac{\|y-x\|^2}{2\sigma^2}}}{e^{-\|x\|}} P(Y = y)$$

Since we're searching for the argmax of this expression regarding x (y is constant so $P(Y = y)$ is constant too).

We have $x^* = \text{argmax}_x e^{-\frac{\|y-x\|^2}{2\sigma^2} + \|x\|}$

since e^{-u} is a monotonic decreasing function.

$$x^* = \text{argmin}_x (-\frac{\|y-x\|^2}{2\sigma^2} + \|x\|) = \text{argmin}_x C(x)$$

Intuitively, the cost function

$$C(x) = -\frac{\|y - x\|^2}{2\sigma^2} + \|x\|$$

is the sum of a:

- data fidelity term (L^2 loss - prediction x shall look like the observation y, relatively to the noise level σ)
- a prior term on the signal (x shall have a small $L1$ norm so it matches best with its prior distribution).

Finding the solution of the cost function

Let's minimize $C(x)$

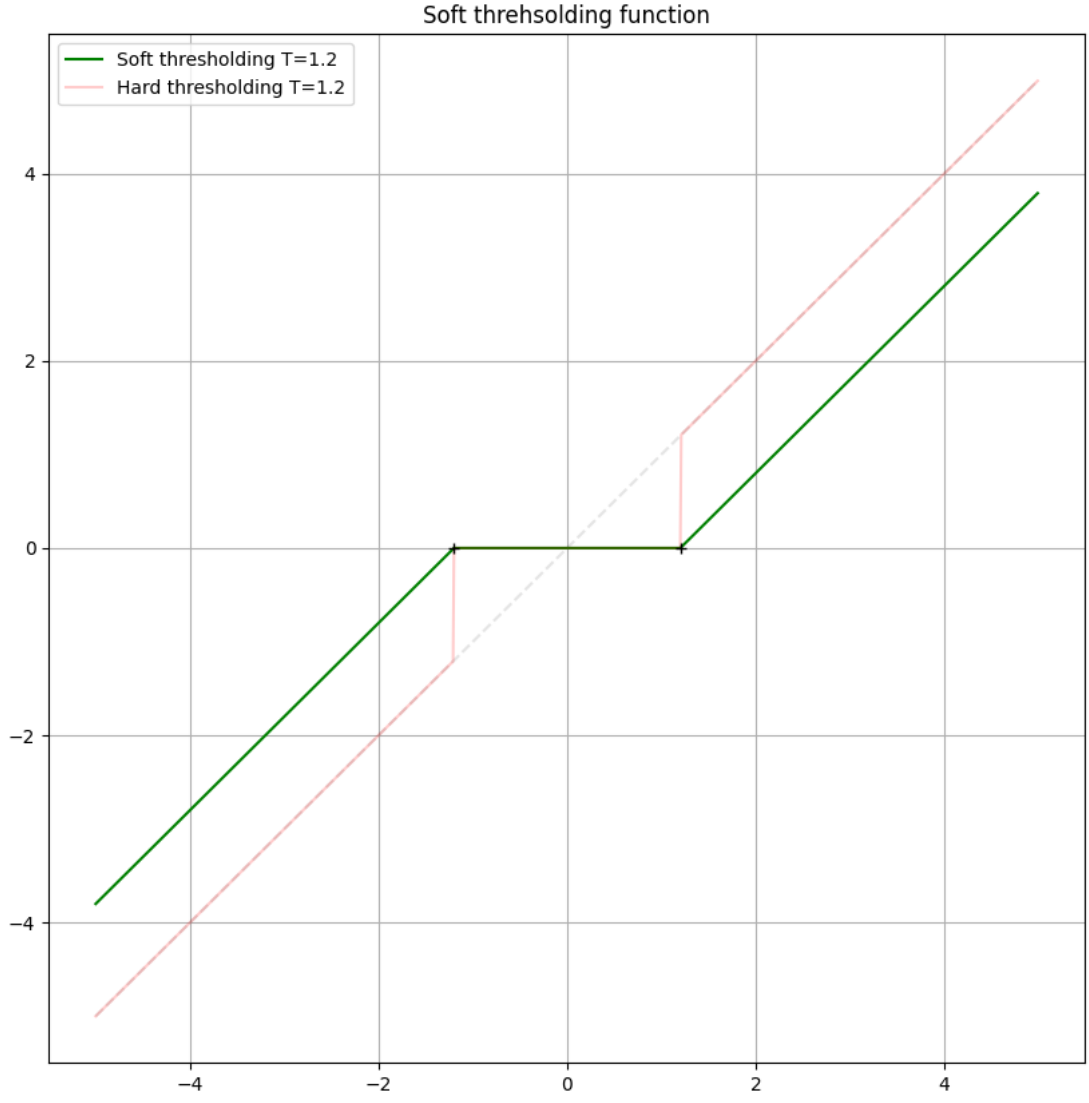
- if $x > 0$, $\frac{dC(x)}{dx} = \frac{x-y}{\sigma^2} + 1$. Critical point shall statisfy $x = y - \sigma^2$
- if $x < 0$, $\frac{dC(x)}{dx} = \frac{x-y}{\sigma^2} - 1$. Critical point shall statisfy $x = y + \sigma^2$

- If $y > \sigma^2$, then $y - \sigma^2 > 0$. Validity of the critical point $x = y - \sigma^2$ ($x > 0$ satisfied).
- If $y < -\sigma^2$, then $y + \sigma^2 < 0$, Validity of the critical point $x = y + \sigma^2$ ($x < 0$ satisfied).
- If $-\sigma^2 \leq y \leq \sigma^2$, the solution might be $x = 0$ as neither of the critical points fall within their respective ranges.

If we check and eliminate wrong solutions, we end up with the following solution called the soft threshold:

$$x^* = \begin{cases} y - \sigma^2 \text{ if } y > \sigma^2 \\ y + \sigma^2 \text{ if } y < -\sigma^2 \\ 0 \text{ if } -\sigma^2 \leq y \leq \sigma^2 \end{cases}$$

This is the **soft thresholding** function.



Please note that one side of this function can be represented by a linear layer with bias and a Relu. Intuitively a CNN (a chain of (conv+Relu)) can learn thresholding in a feature space.

Question 2.

`DCT_denoise` performs a **hard thresholding** in the frequency domain

- applies the DCT transform which is a convolution:
  - by $N * N$ *(number of frequencies)* kernels of size (N, N) .
  - This is performed in the code by `nn.Conv2d` with frozen weights (non learnable).
  - To be an exact DCT transform, one should use a stride of N (shift by a block every time)
- performs a **hard thresholding** ($\neq$ soft thresholding) not in the spatial domain but rather in the frequency domain.
- applies the inverse DCT transform to go back to the spatial domain.
  - This is performed again by a similar non learnable convolution with the iDCT frozen kernels.
  - Since the thresholded result has the same size as the original image (not downsampled by a factor N) - there's redundancy in the spectrum representation and the reconstruction will end up summing NxN iDCT proposal for each pixel (instead of 1)... having NxN overlapping candidate allows reducing blocking artifacts. There's an explicit way to remove these redundant operations have been to use the torch

Minimizing the L2 error in the spatial domain is equivalent to minimizing the L2 error for each frequencies (Parseval theorem).

The assumption for the distribution of X in question 1. is hard to justify in the spatial domain (no reason to have a signal centered at zero with an exponential decay... something similar to the "gray" world assumption). But in the frequency domain, it is more likely to be true (this trick is used in JPEG compression, a lot of image spectrum energy coefficients are close to zero).

Question 3.

- The hard thesholding function can be differentiated with regard to the input y but not with regard to the threshold $T = \sigma^2$ unfortunately.
- A workaround is to use **a very rough approximation** of the hard thresholding function to stay in the standard framework of torch operators: using a bias and a Relu, it is possible to perform an operation of soft thresholding.
- Best idea is to use an differentiable approximation of the hard thresholding function.

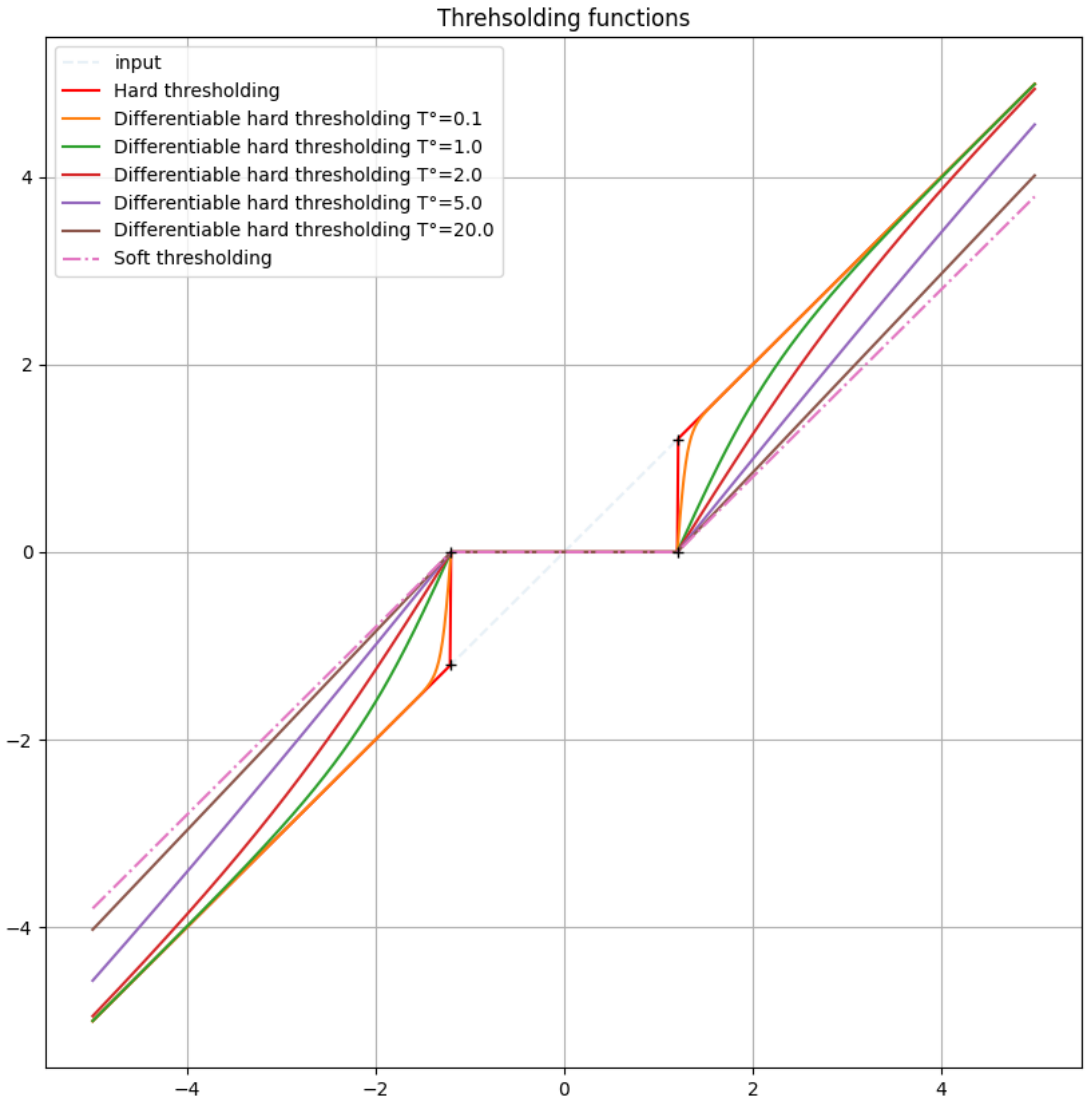
Approximate differentiable hard thresholding function

> The following section is based on the idea I have above and searched if someone did it ... indeed A New Wavelet Thresholding Function Based on Hyperbolic Tangent Function by Can He, Jianchun Xing et al

Another idea is to approximate the hard thresholding by a function satisfying the following properties:

- *differentiable* with regard to the threshold (and the input obviously)
- *parametric*: use a temperature λ parameter so that when the temperature varies from $+\infty$ and 0, the thresholding function varies between a soft threshold and a hard threshold of value T.
- Using this idea, you can use the approximate differentiable function in a deep learning standard framework and proggressively vary the temperature

$$f(x) = \text{ReLU}(x - T + T.tanh(\frac{(x-T)}{\lambda}))$$

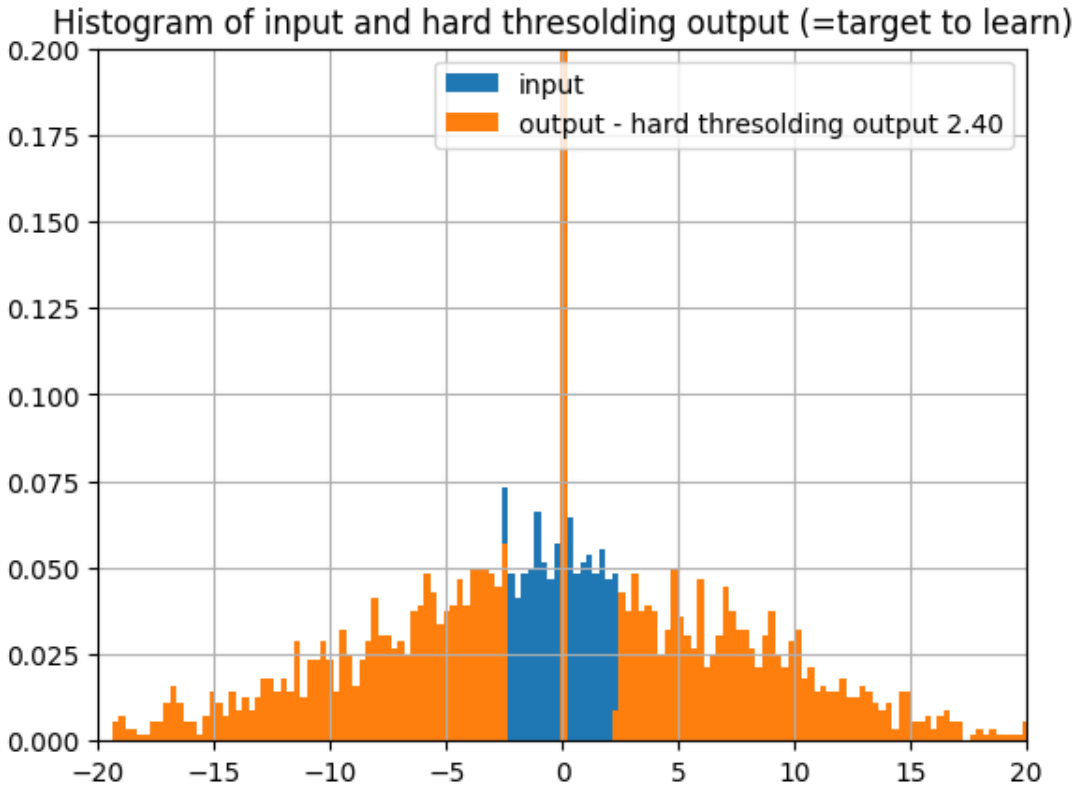



```
# Definition of various thresholding functions
def hard_thresholding(x: torch.Tensor, threshold: torch.Tensor) -
> torch.Tensor:
    """Non-differentiable with regard to threshold"""
    return torch.where(torch.abs(x) > threshold, x,
```

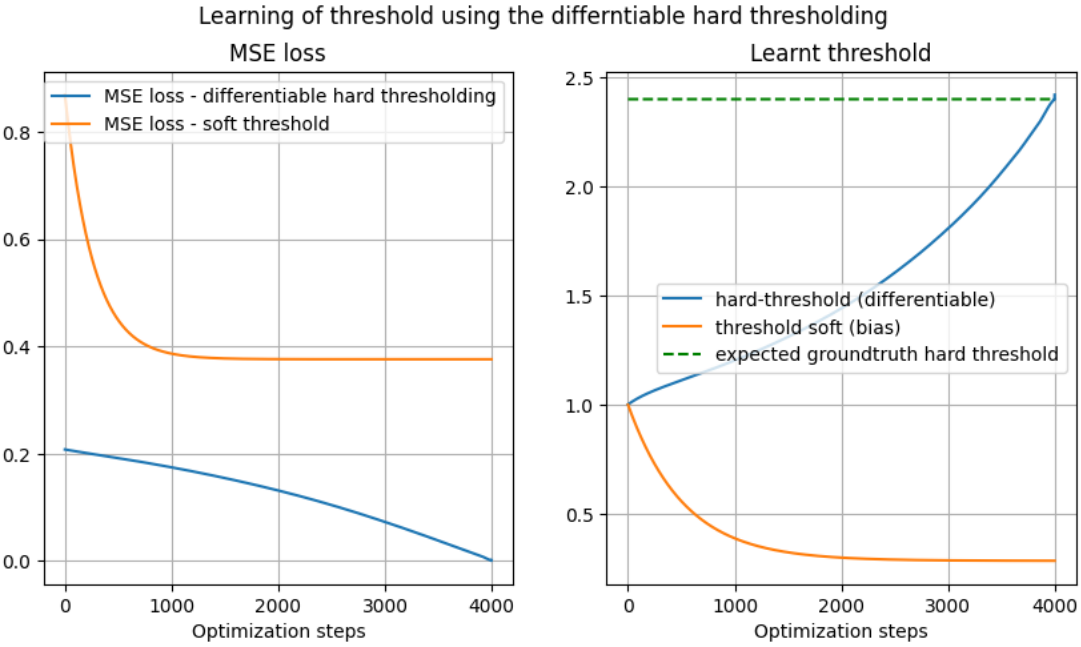
```
torch.zeros_like(x))


def soft_thresholding(x: torch.Tensor, threshold: torch.Tensor) -
> torch.Tensor:
    """Differentiable with regard to threshold, does not preserve
energy of input signal (biased)"""
    return torch.nn.functional.relu(x - threshold) -
torch.nn.functional.relu(-(x + threshold))


def assym_differentiable_hard_thresholding(x: torch.Tensor,
threshold: torch.Tensor, temperature: float=1) -> torch.Tensor:
    x_offset = x - threshold
    return torch.nn.functional.relu(x_offset +
threshold*torch.tanh(x_offset/temperature))


def differentiable_hard_thresholding(x: torch.Tensor, threshold:
torch.Tensor, temperature: float=1) -> torch.Tensor:
    """Approximated of hard thresholding, differentiable with
regard to threshold
    When temperature is high, it is close to soft thresholding
    When temperature is close to 0, it is close to hard
thresholding
    """
    return assym_differentiable_hard_thresholding(x, threshold,
temperature) - assym_differentiable_hard_thresholding(-x,
threshold, temperature)
```

Proof of concept

We build a simple toy example where an input gaussian distribution of standard deviation 8 is hard-thresholded with a trheshold of 2.4.



The goal is to fit/learn this threshold. We'll preform Stochastic gradient descent using the Mean Square Error (L^2) and we'll decrease the temperature progressively.



It is doable to learn the right hard threshold using gradient descent.

Question 4.

The number of significative operations (multiplications) per pixels is in $2 * N^4$

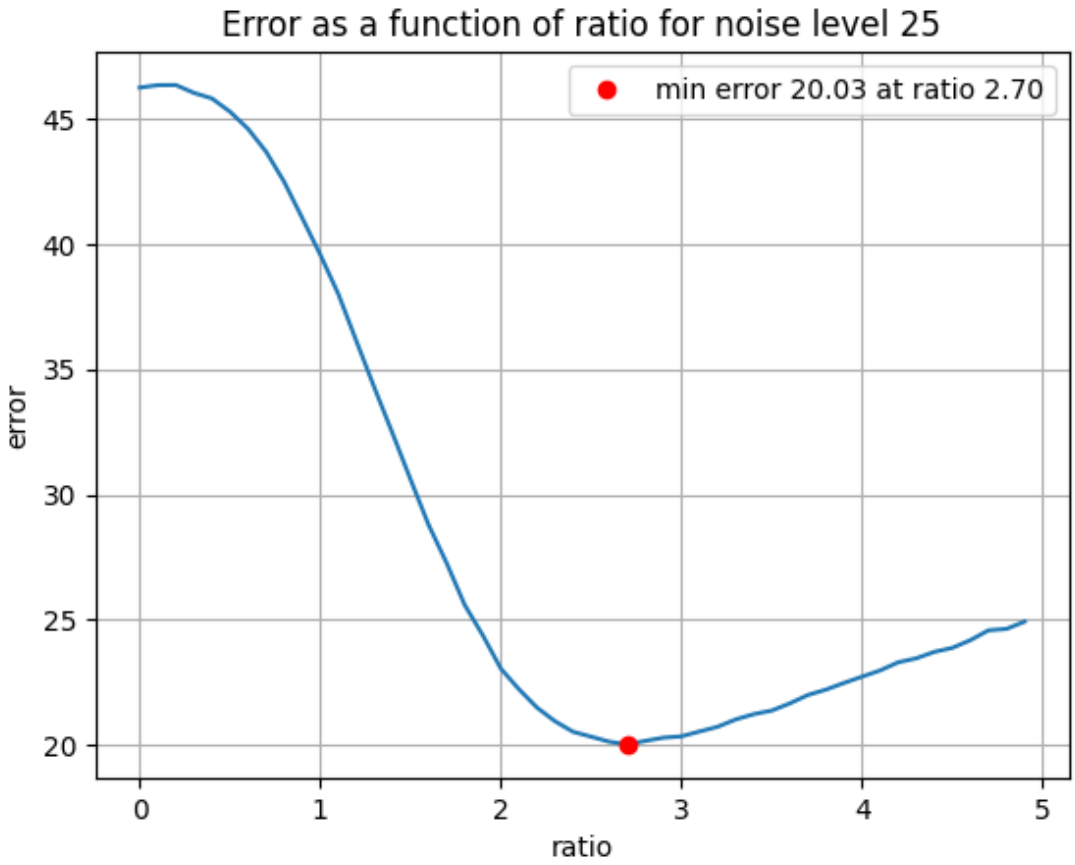
- N^2 frequencies (number of channels) multiplied by
  - convolution kernel of size N^2 multipliciations.

    - 2 because of DCT and inverse DCT.
    - No bias addition, N^2 thresholdings.
- Final normalization is negligible.

Question 5.

Best threshold (ratio) value for the Zebre picture with AWGN $\sigma = 25$ is 2.7.

$r = \frac{T}{\sigma} = 2.7$



Noisy	**DCT denoised** $r = 2.7$	**DCT denoised** $r = 5$
		
$\sigma = 25$	Optimum DCT denoiser in the MSE sense	Oversmoothed, threshold is too high

FFDNET

Question 6

High noise level conditions: $\sigma_{255} = 50$

Type	Noisy	DCT denoised	FFDNET
			
Details	$\sigma = 50$	Optimum DCT denoiser in the MSE sense *optimized tuning* $r = 2.9$	FFDNET
Residual's standard deviation (/255) *lower is better*	50.11	15.61	13.24
PSNR (db) *higher is better*	14.1	24.2	25.7

Mild noise level conditions: $\sigma_{255} = 25$

Type	Noisy	DCT denoised	FFDNET
			
Details	$\sigma = 25$	Optimum DCT denoiser in the MSE sense *optimized tuning* $r = 2.8$	FFDNET
Residual's standard deviation (/255) *lower is better*	25.01	10.87	9.82
PSNR (db) *higher is better*	20.16	27.39	28.28

FFDNet creates much sharper and less noisy images. DCT denoiser leaves structured patterns un flat areas (*tuning of the DCT denoiser could probably be pushed further with spectral threhsolds depending on the frequency and maybe increasing the filters size). We clearly see here the advantage of using a deep convolutional network over spectral thresholding. We also nottice some weird patterns in the grass next to the zebras legs (activation function which triggers where they should not and start hallucinating structure content among the noise). The visual improvement of FFDNet over DCT denoiser is confirmed by the metrics ($\mathrm{PSNR}_{\mathrm{FFDNET}} > \mathrm{PSNR}_{\mathrm{DCT\ denoiser}}$).

Question 7

By counting the number of parameters instance, we can get an estimation of the number of operations per pixels, here $4.86 * 10^5$
This is done by using the `torch.numel` methode to the FFDNet model instance

code: `sum(p.numel() for p in ffdnet.parameters()) # >>> 486080`

In depth analyzis

In the IPOL paper, we read that the network has $W = 64$, $K = 3$ and depth $D = 15$, leading to an overall **rough number of convolution coefficients** of $D * K^2.W^2 = 15 * 3 * 3 * 64 * 64 = 55960$ which they report as $5.6.10^5$ parameters which is not correct. In the model `IntermediateDnCNN` line `FFDNET/models.py` line 50, we see that the first and last layers are smaller.

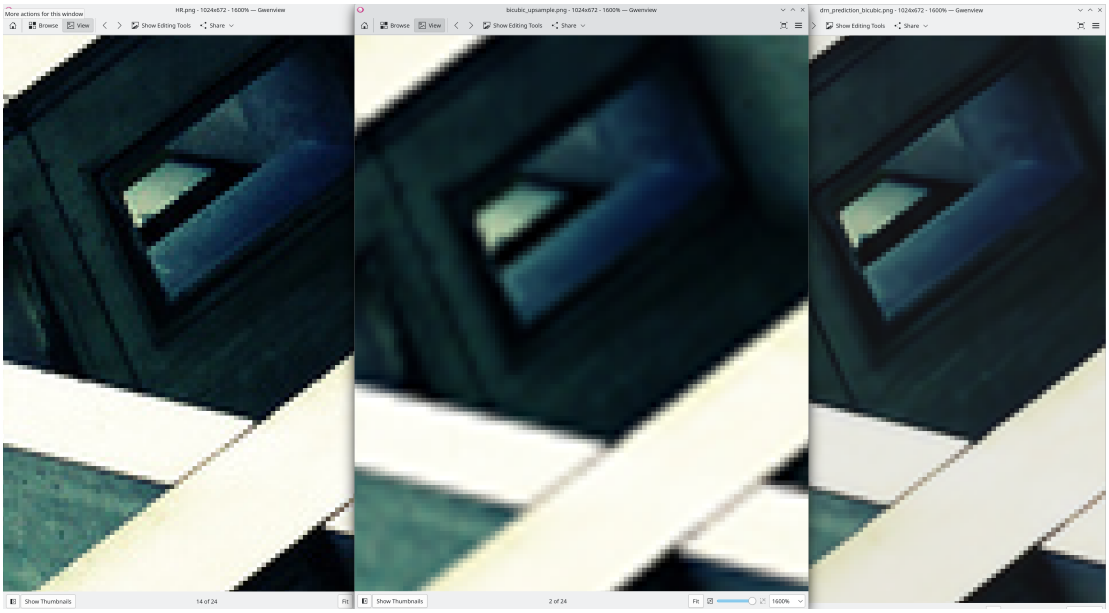
$(D-2) * K^2.W^2 + 2 * K^2.W * (C=1) = 13 * 3 * 3 * 64 * 64 + 2 * 3 * 3 * 64 =$
This is much closer to what we get from the torch provided number (we ommited the biases and batch norm coefficients).

The trick of decimating the image and perform 4 times the network at half resolution does not affect the number of operations per output pixels but it reduces the memory footprint (and computation cost according to the authors) but most notably, it allows the denoiser to enlarge its receptive field.

DRN

At 1600% to view details correctly.

- Left: original HR (high resolution)
- Middle: upsampled degraded LR (low resolution)
- Right: Network SR prediction, a lot of details have been recovered. (this is impressive although very very far from real photography conditions.)



Question 8

RDN is made of 2 main parts.

- 1st part works at low resolution (but no extra poolings) and is made of a combination of multiple convolutional base blocks. As a common practice in machine learning, authors propose a building block and make combinations of it.
  - 3x3 convolutional + ReLU (non linear activation) blocks . They increase connectivity among the layers of the base block by introducing a bunch of residual connections.
  - The addition of multiple residuals connections allows different depth to communicate information...
  - The proposed "feature fusion" is an extension of the residual connections proposed in ResNet. Instead of an addition of the previous layers with the current layers, they propose to concatenate the tensors and leave a linear layer mix them with a pointwise convolution (a **conv 1x1** mixes the channels together without the spatial aspect of a convolution.).
- 2nd part is the actual upscaler (*The upsample is in charge of producing the missing information.*). High resolution information has been stacked aggregated among the channels rather than on a the spatial dimension. This is done using the torch.nn.PixelShuffle which was introduced in Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network by Shi et. al (2016) and mentioned in the RDN paper as "ESPCN" block. Performing most operations in the low resolution space reduces the computation cost and memory requirements. But in the RDN paper, they still use a final convolution working at high resolution.

The network has been trained in a supervised fashion. Supervision comes from the colored ground truth high resolution (HR) image taken from the popular high quality image dataset named Div2K (in practice training is performed on tiny crops

to create batches). Low resolution image is generated from the high resolution (HR) groundtruth image.

Below is the training scheme: Trainable super resolution network in orange, image buffers in blue, degradation operator is either the "Matlab bicubic downsample" or a gaussian blur followed by a decimation.

Question 9

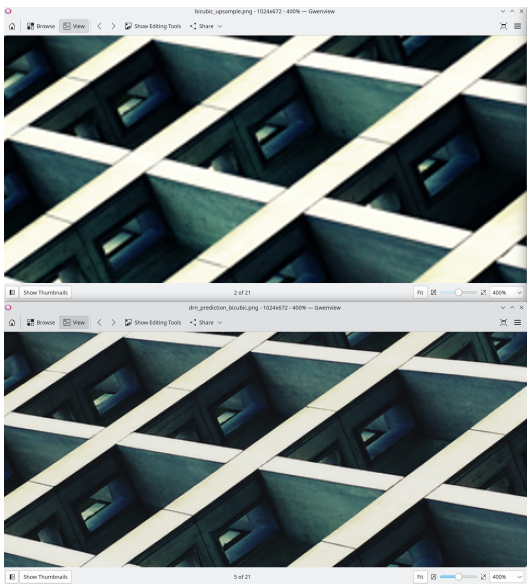
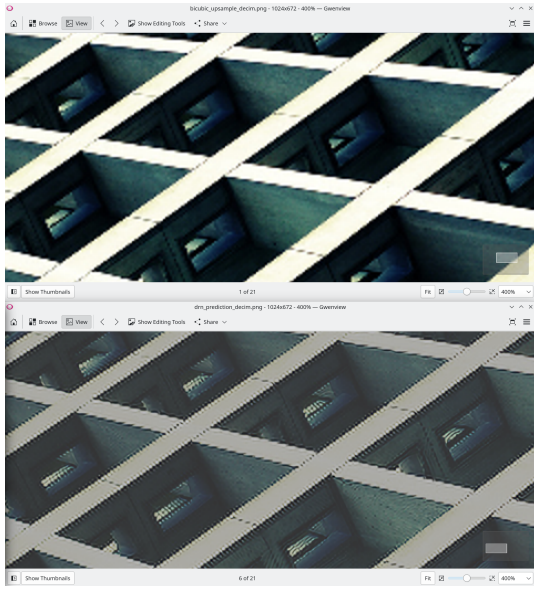
The DRN network used **22.1M (million)** parameters which is a rough approximation of the number of operations per pixel.

Same trick used as in question 7. Let's count the number of parameters to get a rough approximation of the number of operations per pixel.

```
sum(p.numel() for p in model2.parameters())  # >>22123395
```

Question 10

- Top: bicubic upscale of the low resolution image
- Bottom: DRN network inference performed on the low resolution image.

Degradation ="bicubic" downsample	Degradation: decimation (skip)
	

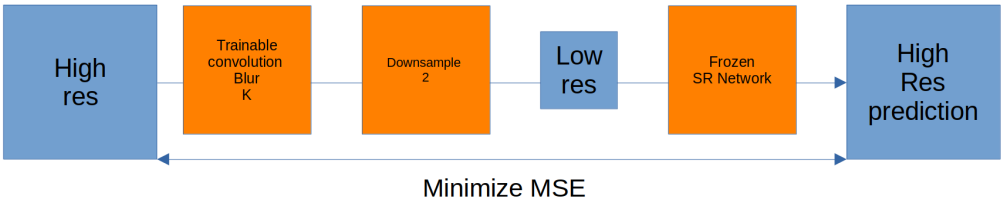
Degradation ="bicubic" downsample	Degradation: decimation (skip)
DRN works in its training condition regime. Results are impressive. DRN was able to recover the details, at least the super resolved images look much sharper than the bicubic upscaler at the top.	DRN inference performed on data out of the training distribution (the applied degradation operator does not match with the so called bicubic used during training). Performances are seriously impacted and aliasing it critically amplified. Another notticeable effect is that the network has added a kind of additive offset... impacting the low frequency is quite visible.

Conclusion Do not use a SR network out of it's training conditions. The gap between training and real conditions may result in significant artifacts.

Question 11

To find the degradation operator, one can try to estimate a blur operator by a tricky gradient descent scheme:

- try to minimize the L^2 loss between the prediction and the groundtruth error.
- let the blur kernel as a parameter (the one which is optimized)
- super resolution network's weights are frozen (*but thanks to auto differentiation, gradients can still flow through this.*) Below is a scheme for this idea.



- (`torch.nn.Module` s) operators in orange
- Images in blue

One could perform this descent on a single or multiple images. The goal is to find the degradation that fits best the degradation operation which was used during training.

> Please note that if the operator is non linear, tiny CNN could replace the linear blur operator.

> Fitting even a linear kernel correctly is not that easy, some researchers used several linear layers (yes! a linear network with no non linearity) in Kernel GAN to ease this search.

The other approach is to read the paper and sometimes the provided source code.