14.12.22

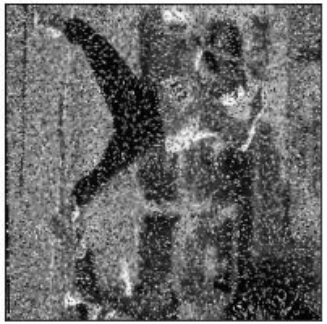
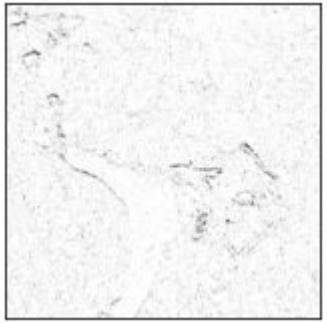
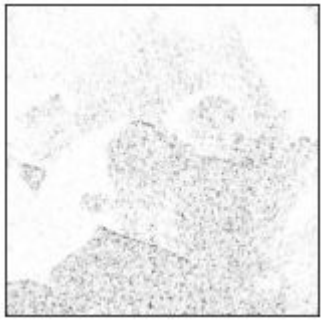
Performance while learning lambdas

Brief description of performance and color code

As I said before, whereas the classic implementation of the anisotropic diffusion didn’t improve, it allowed the networks to boost their performance, which can be seen in both their results and the characteristics the coefficient functions where getting because of the chosen lambda.

One thing that I found interesting is that now the coefficient functions show different strategies depending on the noise and the image. For instance, they were able to distinguish between regions based on their intensity, favoring a high diffusion on darker areas while preventing it in lighter ones when dealing Gaussian and Poisson noises, but failing to do so for the s&p noise. On the next slide, the whiter the color, the bigger the value for the coefficient function.

Examples of diffusion in dark areas

	Gaussian noise	Poisson noise	Salt and pepper
Noisy image			
Coefficient function			

Other strategies

Some other times, what the networks were doing was focusing on finding the borders from the images and preventing the diffusion to take place across them, specially for the deblurring and inpainting case, which is a natural strategy to think about for this type of noises.

Moreover, I also found this behavior in other types of noise when the variance was small, something which again is natural to expect since the network job basically becomes stopping diffusion across borders.

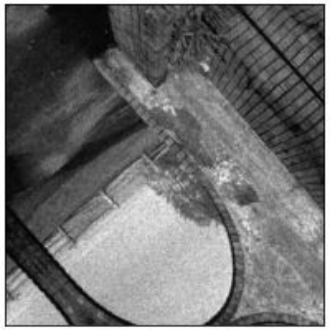
Examples can be found in the next slide where the first row corresponds to noisy images and the second to coefficient functions.

Examples of border preservation

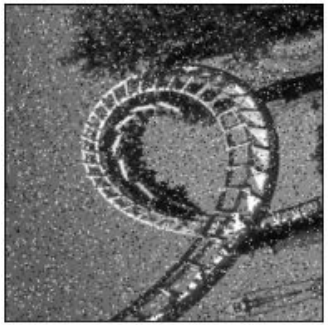
Deblurring

Gaussian noise

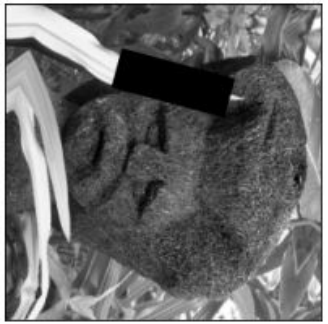
Poisson noise

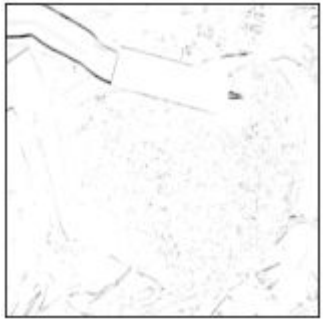


Salt and pepper



Inpainting





Observations on the coefficient functions

Although a natural strategy was being displayed by the networks when it comes to the coefficient functions, the difference between its high and low values was pretty mild, leading me to believe that this architecture wasn’t being able to do all it would have liked to do.

In addition, I didn’t notice a big difference on the behavior nor on the strategies the networks were taking when I was changing between function 1 and 2.

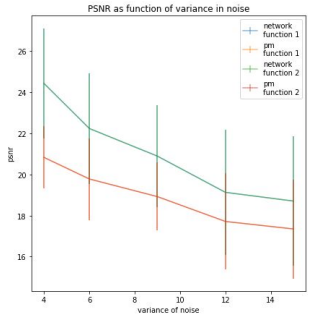
Networks vs. classical implementation

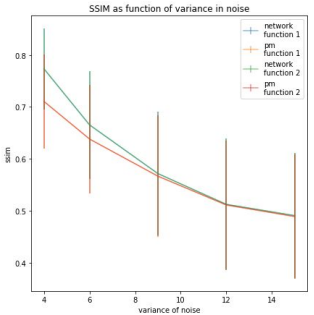
As I said before, the modifications made on my previous implementation allowed the networks to improve and beat the classical implementation in almost all the cases I considered but for the salt and pepper noise. Together with what I've shown before, this makes me think that the ability to recognize intensities and important borders in the image play an important role when using this denoising technique.

You can see a comparison of the performances on the next slides. They were made averaging over 100 generated images per noise level I considered.

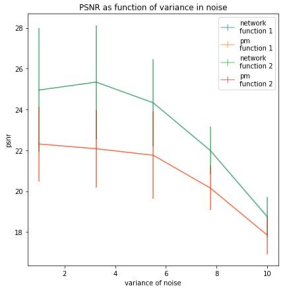
Networks vs. classical implementation (PSNR and SSIM)

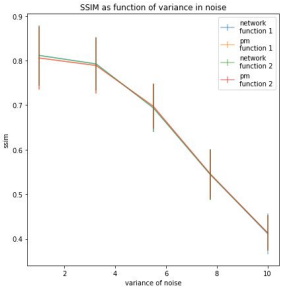
Deblurring



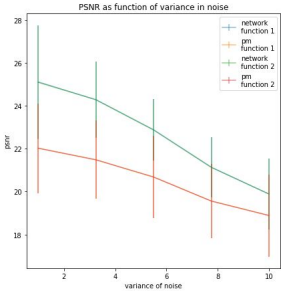


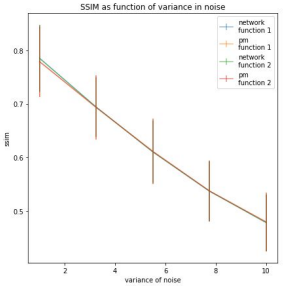
Gaussian noise



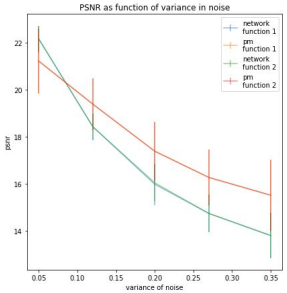


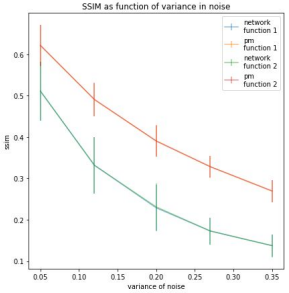
Poisson noise



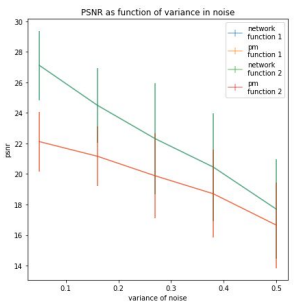


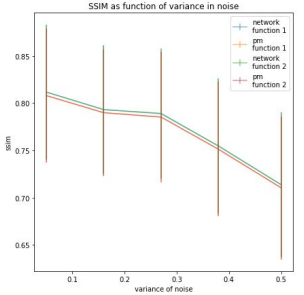
Salt and pepper





Inpainting





Performance of P1 functions

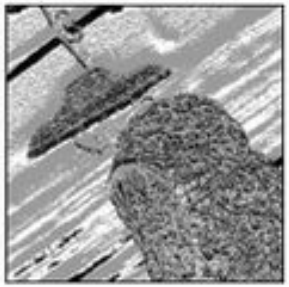
Description of implementation

The way I implemented this P1 functions was by making a partition from 0 to the maximum value the image's gradient norm reached and fitting an affine function in each one of the parts.

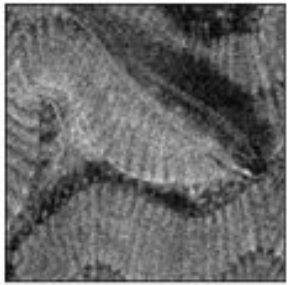
Following the lambda-learning case, I first took a look at the coefficient functions the networks produced. It can be seen that for each type of noise, the network was distinguishing smooth regions on the image and deciding how much it should be diffused. Some examples can be found on the next slide where the first row corresponds to noisy images and the second to coefficient functions.

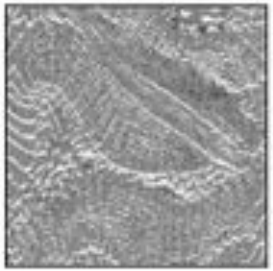
Examples of coefficient functions

Deblurring

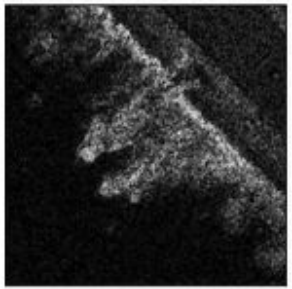


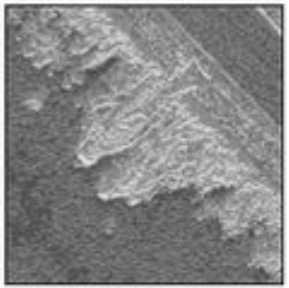
Gaussian noise



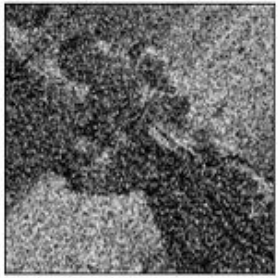


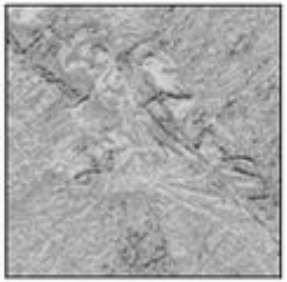
Poisson noise



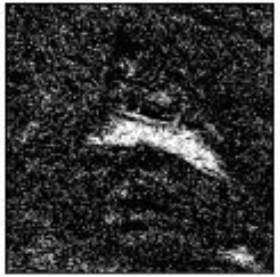


Salt and pepper





Inpainting



Discussion about P1 function’s performance

Besides the fact that this type of functions have the ability of producing negative values for the coefficients of the spatial derivative, it’s very clear that they can produce much richer and continuous coefficient functions which more or less segment the image distinguishing connected regions in which the image behaves smoothly.

It’s also remarkable that they can identify the part of the image which needs to be inpainted without any specification within the architecture, ability which can be noted in the way it produces the coefficient functions for this type of noise.

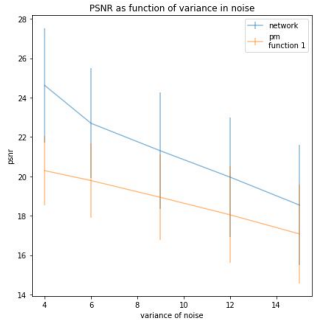
Discussion about P1 function’s performance

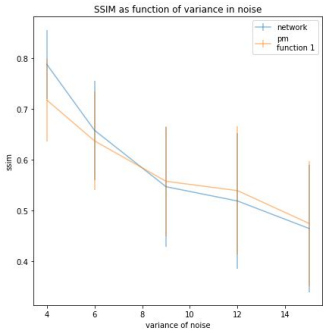
This feature of segmenting the image can be explained by the convolutional layers involved in the definition of the P1 functions. In order for the networks to have an idea of what’s happening around the pixel they’re considering and therefore being able to distinguish better noise from border, I added some convolutional layers.

Thus, while those layers allow the network to know which borders it should preserve, they also force it not to have a pointwise behavior. This tradeoff has pros and cons which can be seen in the plot on the next slide.

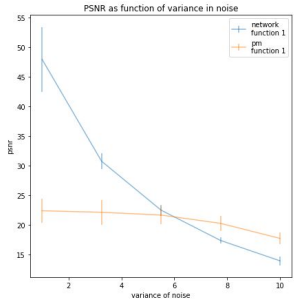
P1 functions performance vs. anisotropic diffusion

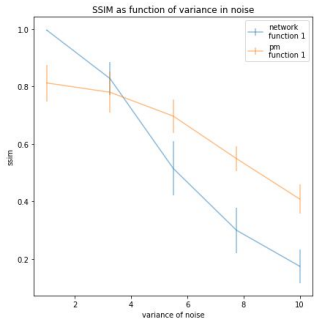
Deblurring



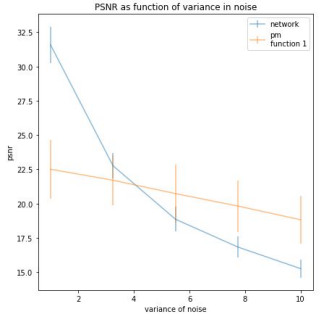


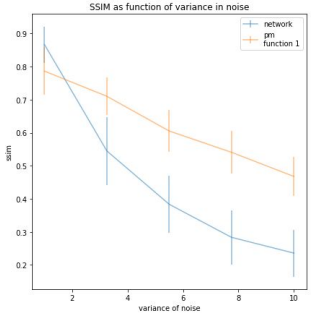
Gaussian noise



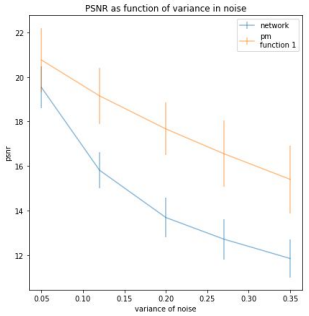


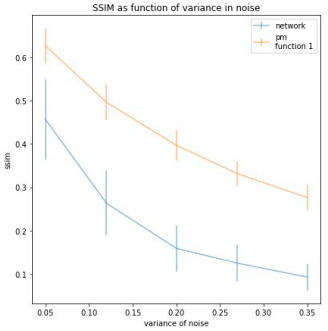
Poisson noise



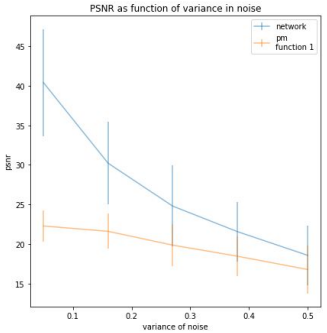


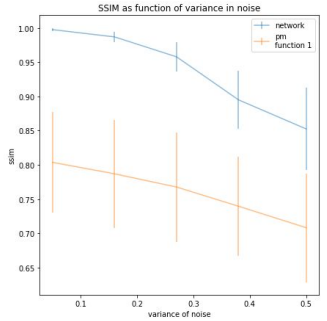
Salt and pepper





Inpainting





P1 functions performance vs. anisotropic diffusion

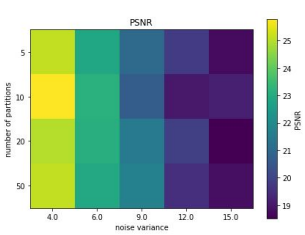
It can be seen how, leaving s&p aside, the P1 functions outperformed the classic anisotropic diffusion for small variances while getting worse as the variance grows. This can be understood for what I've said before: this type of functions are doing a great job at distinguishing regions of the image the should preserve, but they don't know how to deal with the local disturbances generated by the noise.

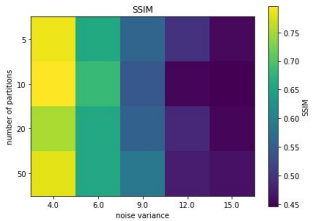
This also shed some light on the s&p case since this type of noise characterizes for destroying the regions this type of functions know how to deal with.

I show the next slides to get an idea on the impact the number of partitions has on the network performance.

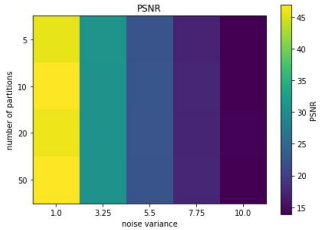
Effects of modifying the partition

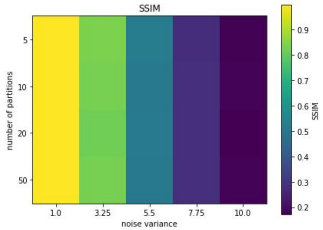
Deblurring



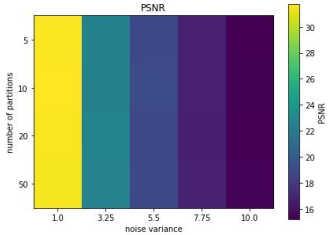


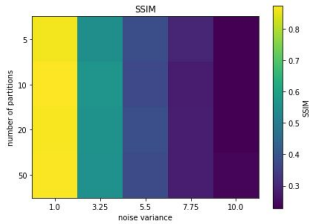
Gaussian noise



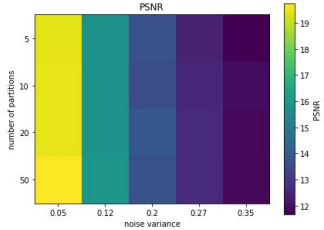


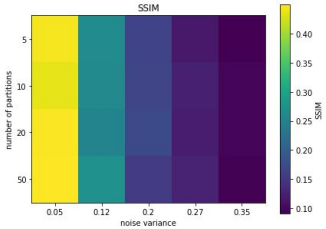
Poisson noise



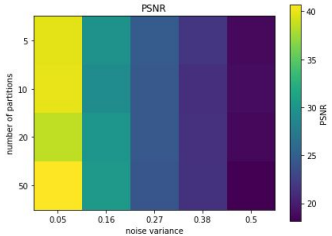


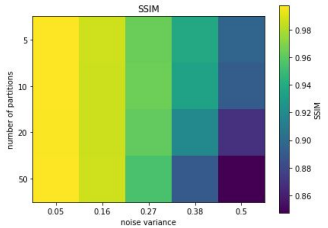
Salt and pepper





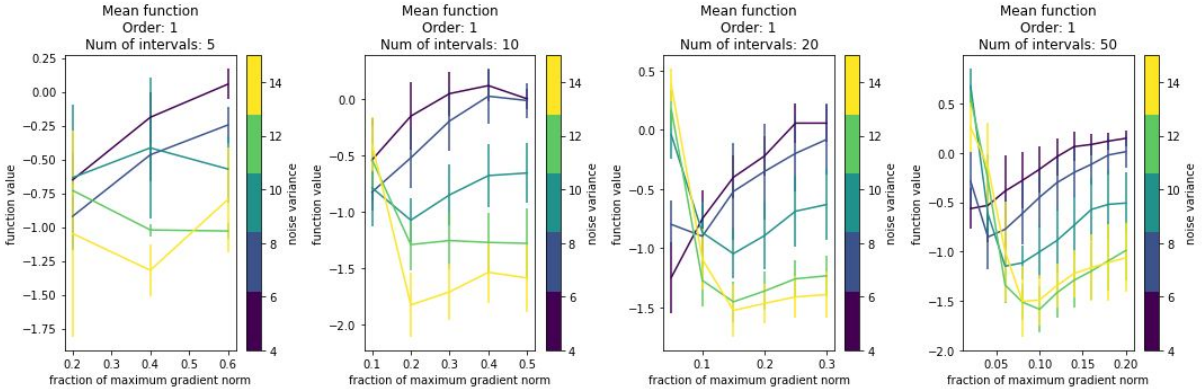
Inpainting



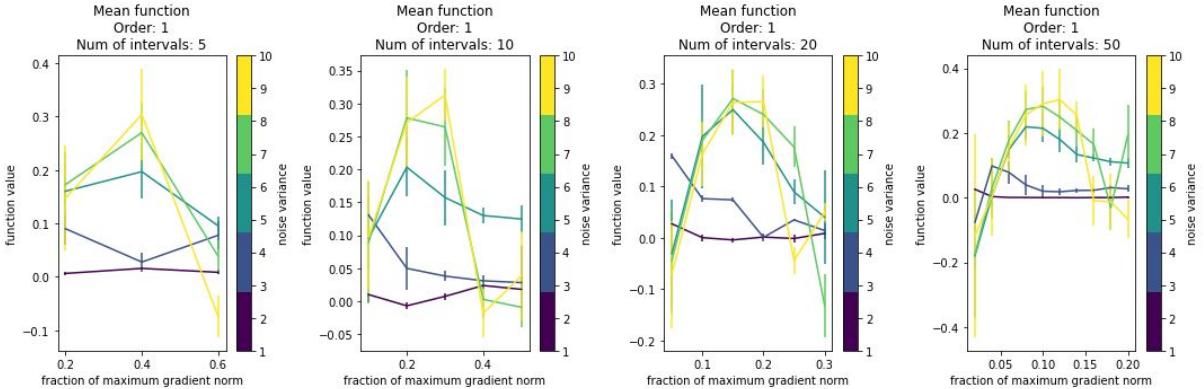


Effects of modifying the partition on the P1 function

Deblurring



Gaussian noise



Effects of modifying the partition

We can see from the previous heatmaps how the effect the cardinality of the partition has on the performance of the denoiser depends a lot on the type of noise we are dealing with.

On the other hand, from the plots on the previous slide we can see how the finer the partition is, the better it fits some given function, where it changes with the noise type and its variance. This behavior was also seen on the other type of noises I’ve been considering.

Edginess

Motivation

As I told you on a previous email, one problem I was having with this model was that the networks were not being able to identify which borders were the ones they should preserve and which ones were to eliminate for them being caused by the noise addition, but to talk with more precision about this, I needed some measure of the “edginess” an image has.

To do so, I defined the edginess of an image as the integral of the absolute value of its divergence, which in this discrete scenario I computed as just the sum of the absolute value of the discrete directional derivatives computed over each pixel.

Motivation

There are a couple of ideas behind this way of measuring the edginess of an image. One of them is just because on doing so, I would be able to take into account how abrupt were the changes of intensity across the image.

On the other hand, understanding the image as a piecewise smooth function with the non smooth parts in which there's a big change of intensity, i.e. original edges I would like to preserve, I would be integrating, grace to the divergence theorem, an unitary function over every edge of the image, once again summing up in some way the amount of boundaries I originally had.

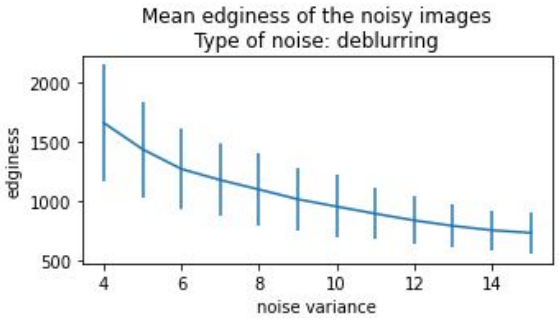
Effect of variance of noise on image's edginess

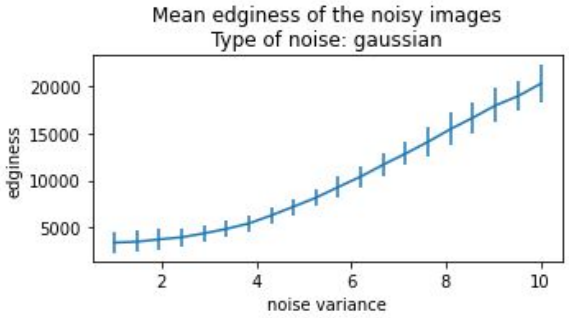
As discussed before, what I'm expecting is that the bigger the variance for the noise, the bigger the edginess of the noisy image when considering Gaussian, Poisson and s&p noises. On the other hand, I would expect that the bigger the kernel for the blur, the smaller the image edginess. In contrast, for the case of the inpainting, I am creating a new border by erasing all information in certain area, but at the same time I'm erasing every border inside of the rectangle. Thus, the edginess shouldn't change a lot for this type of noise.

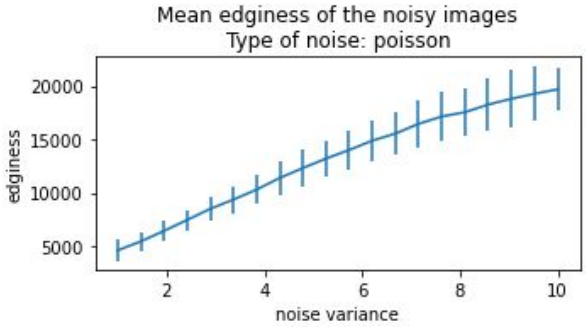
This suppositions can be mathematically proven using the definition of the different noises.

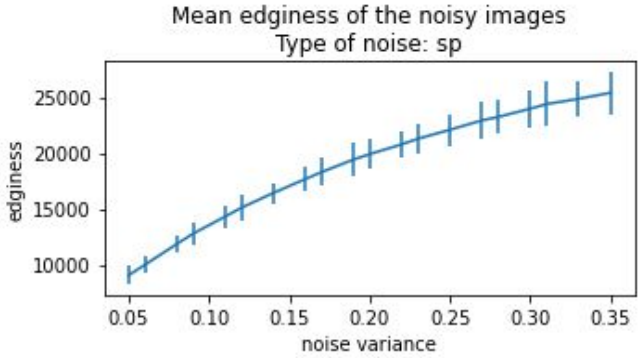
On the next slide, you can find the mean edginess as a function of noise's variance.

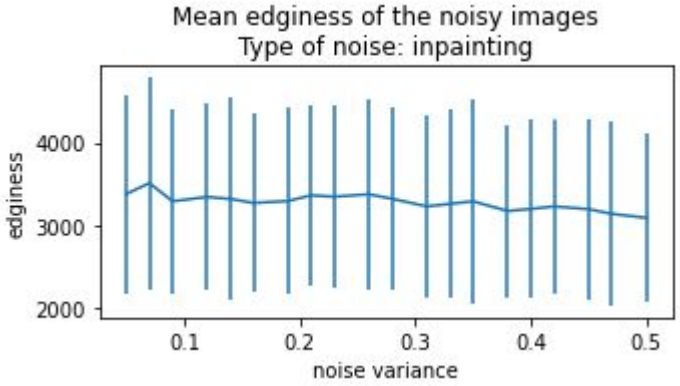
Edginess as function of noise variance







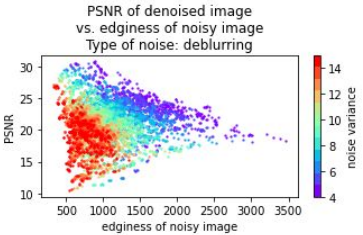


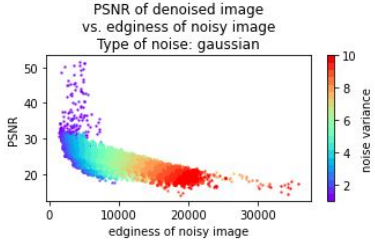


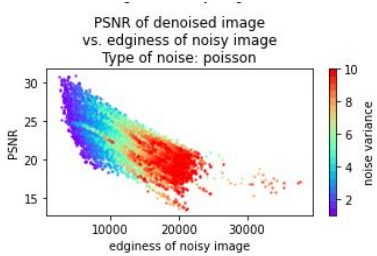
Relation between edginess and PSNR

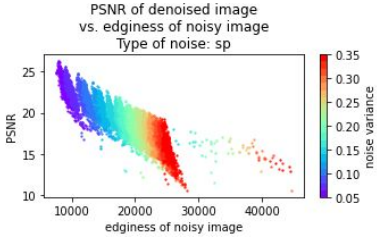
In the next slide you can find some plots showing how it is true that the bigger the edginess of an image, the worse this model will behave, which can be understood since even if it is known for preserving edges, this type of anisotropic diffusion is at the end of the day a heat equation trying to homogenize all the “energy”, i.e. the intensity, in the image.

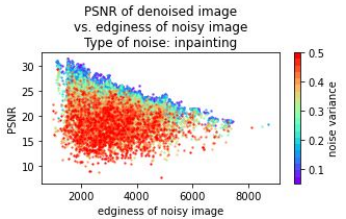
Relation between edginess and PSNR

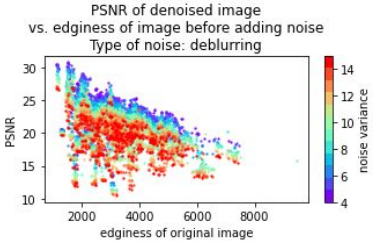


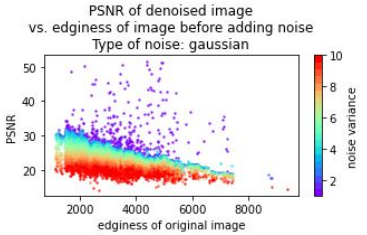


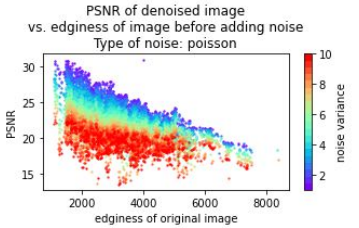


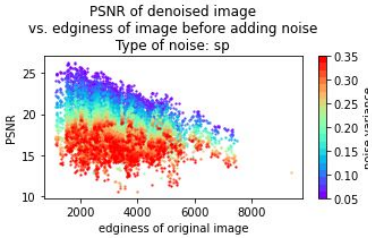


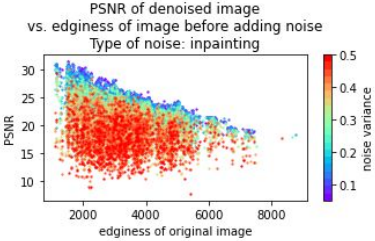












Discussion

We can see in the previous slide that this way of measuring the amount of edges an image has, actually reveals a tendency relating the PSNR and the edginess, which depends on the type of noise we're dealing with.

Because of this, I thought it would be helpful for the denoising task to give the network the capacity to locate the edges I want to be preserved.

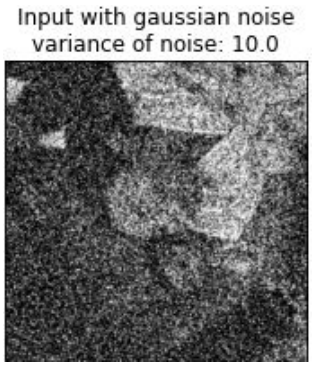
Border classification

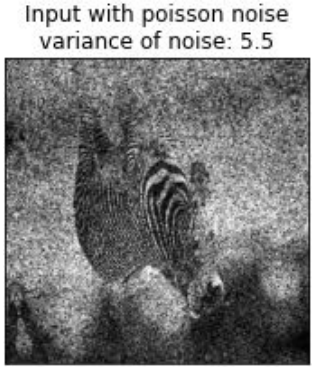
Training networks to locate borders

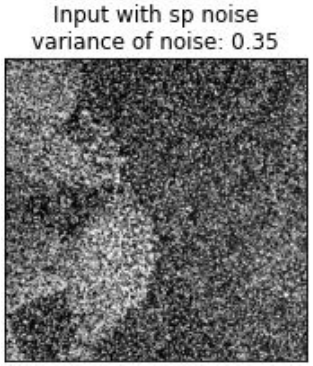
For this task, I created some data generators which would take compute the infinite norm of the gradient for each pixel in the image and consider a border those points in which it was greater than certain threshold.

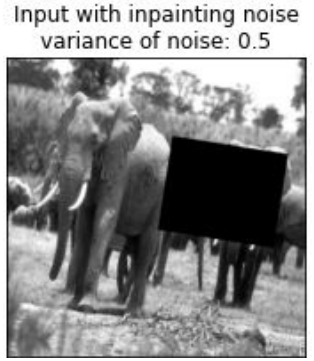
As for the network characteristics, I used a UNet with around 100,000 parameters with a sigmoid activation function on the output layer, taking as an input a noisy image. You can see some instances of the performance on the next slide.

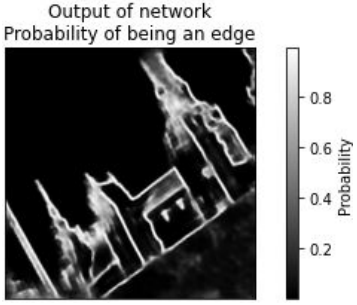
Examples of border location

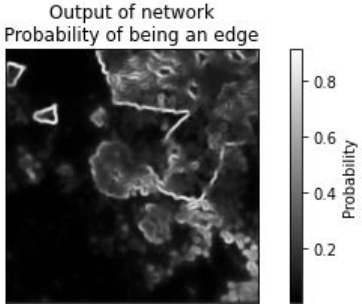


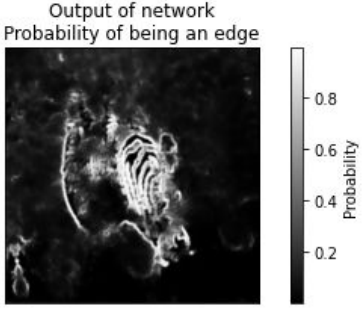


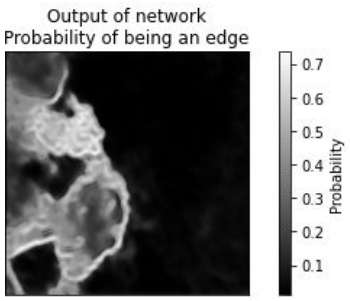


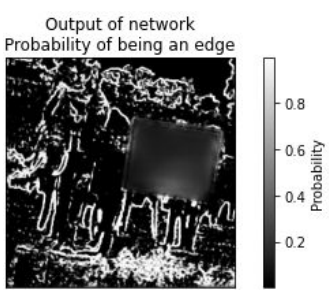












Discussion

With the previous examples it can be seen that it is possible to detect the actual borders of the original image using the noisy one as an input for the neural network. Moreover, since there are only convolutional layers, this can be used on images of any size.

Once I had this, I trained the previous models again, but now generating one lambda value for borders and another for not borders, doing the corresponding thing with the P1 functions. While this training was taking place, I froze the weights of the border classifier.

Learning lambdas with border classification

Description of implementation

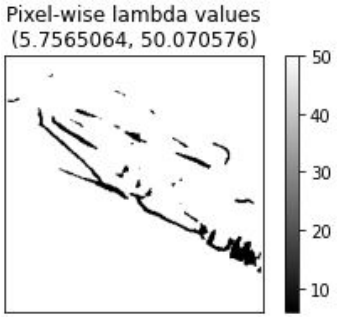
As I said before, I computed two lambdas this time: one for the pixels identified as borders and one for the identified as part of a smooth region.

I show how the lambdas were assigned to each pixel for the different noises in the next slide.

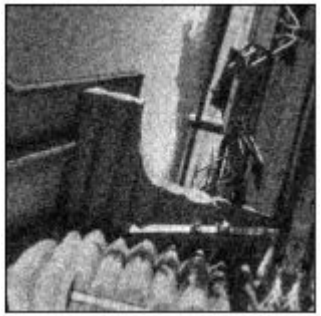
As said before, the model I trained in these cases were the ones I had before and unless specified, the used function was the option 1, i.e. exponential with negative argument.

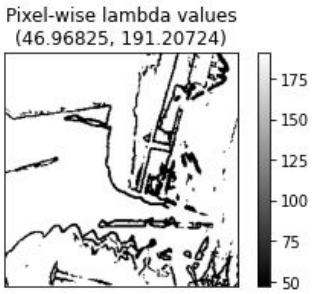
Pixelwise lambdas assignment

Deblurring

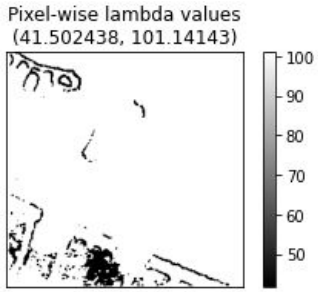


Gaussian noise

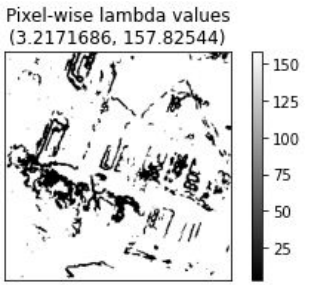




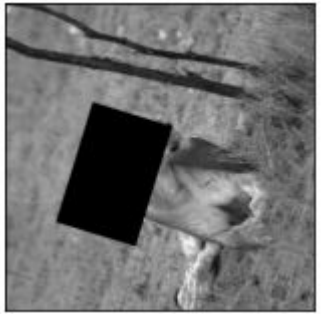
Poisson noise

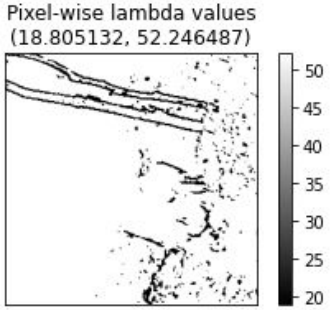


Salt and pepper



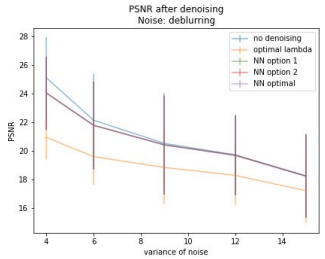
Inpainting

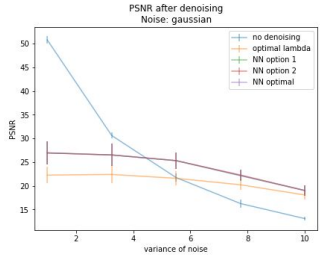


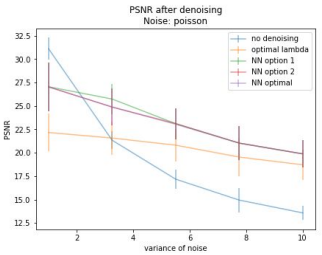


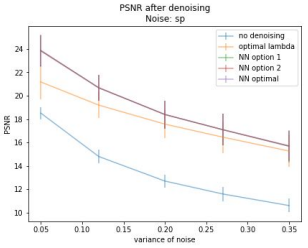
Performance comparison

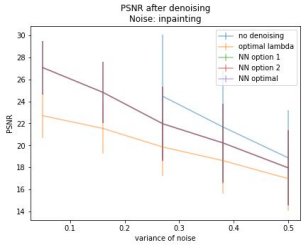
Deblurring | Gaussian noise | Poisson noise | Salt and pepper | Inpainting

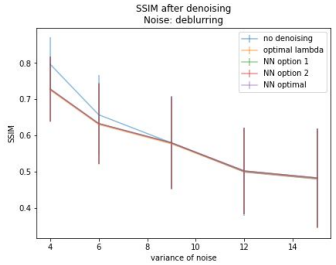


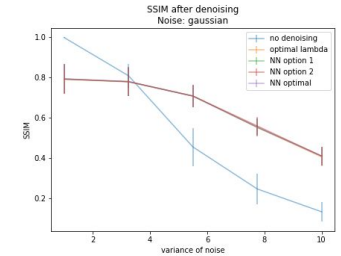


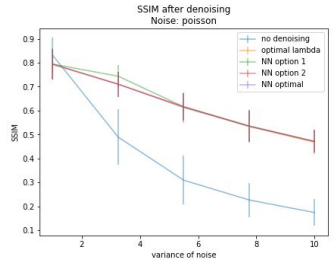


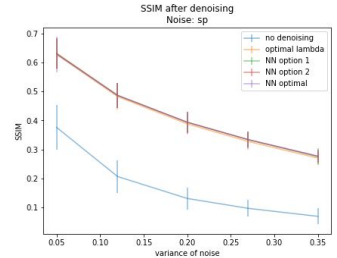


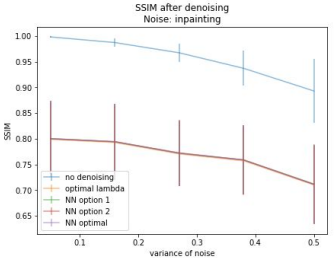












Observations and discussion

We can see how the networks are recognizing the need to have very different values for the lambdas depending on the classification for the pixel type, but also how this method didn’t achieve a significantly better performance compared to the case in which only one lambda is assigned to the whole image neither for the PSNR nor the SSIM.

P1 functions with border classification

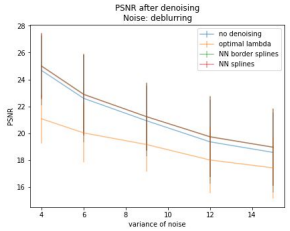
Description of implementation

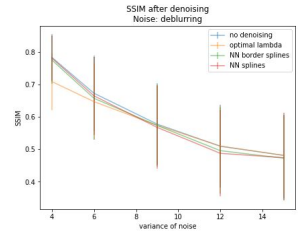
Afterwards, I used the same network as before to detect the edges on the image and then use the previously used P1-functions estimator to apply the anisotropic diffusion.

A comparison for the performances of this case can be found on the next slide.

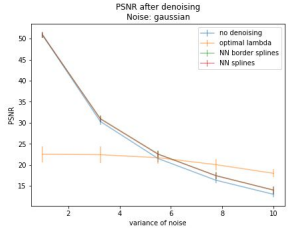
Performance comparison

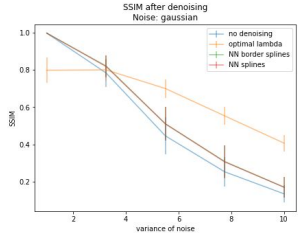
Deblurring



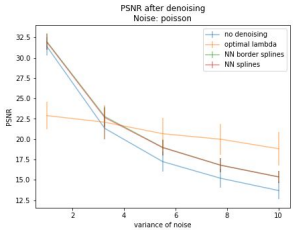


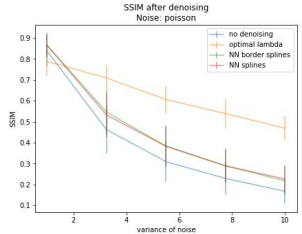
Gaussian noise



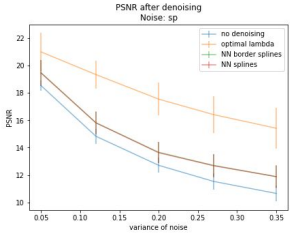


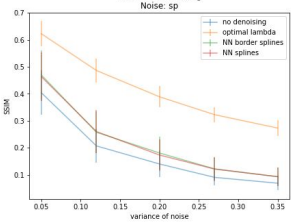
Poisson noise



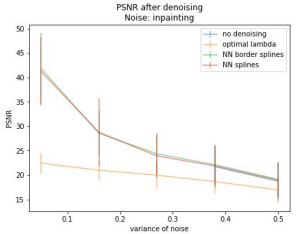


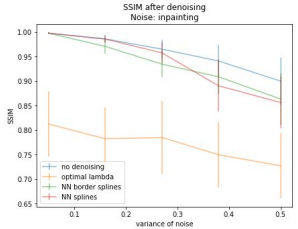
Salt and pepper





Inpainting





Discussion on performance

In contrast with the previous case in which I used a given function taking a lambda value as part of the argument, we can see a noticeable difference between the performances of the networks which were capable of distinguishing borders and not, at least for the SSIM.

Nevertheless, improvement wasn’t significant.

Further discussion

Discussion

By their properties, the way I implemented this networks gives us information not only about piecewise affine functions' performance but also about the one corresponding to piecewise C1 functions, being that the reason I didn't implement splines of higher orders.

On the other hand, the next step Alexander had proposed to me on our call was to use decreasing functions, which I was planning to implement by using simple decreasing functions. Nevertheless, by the results I'm getting from the P1 functions and their non-monotone nature (slide 26), this technique would only put and extra constraint and I don't know how worthy it is.

Discussion

Looking at the results obtained from the different techniques shown here, it can be seen that the neural networks outperformed the method in which I just picked the mean optimal lambda and applied it to all the data set.

Despite this, the plots and results shown in this presentation hint that there are several modifications that can be made in order to achieve better results. In the next section, I'll present some of the models I believe are the most promising.

Proposals for future work

Adversarial networks

GAN literature has proven that letting two architectures compete as their training method may lead to better results for both of them, technique which has been specially used in image-related tasks.

With this in mind and thinking of this denoising tasks as generating an image given a random one, I would like to implement a generative network except that the generator would be a denoiser (e.g. the one I already implemented, but not restricted to it) and for the discriminator I could use a siamese network with triplet loss function like in [1], couple which has shown good results while dealing with the *Totally looks like* data set.

Adversarial networks

This discriminator could be the one used on [1], composed by an encoder and a distance layer, where the later would be just the mean squared error.

One upside that I can see with this is that I would stay in the pixel space, so I would be able to see exactly what's going on, while at the same time make the distinction on a latent space, where literature has shown better performance.

Example of Totally looks like data set

Denoising diffusion implicit models

Another technique which looks promising is the one found in [2] and implemented in [3]. Here, people see the diffusion process as a Markov chain on a latent space (which may include an stochastic part as in [4] but that is usually used to have more variety while generating results) going from an image to pure noise, allowing them use the inverse chain to obtain images out of nothing but random values.

The most common latent space used in this cases is a real version of the Fourier transform and they usually work in subspaces of the unit sphere claiming they get better results this way compared against when they don’t impose this restriction (I wonder if this can be explained by a compactness property of the sphere).

Denoising diffusion implicit models

The idea here would be again to train this as a generator and use the discriminator to make sure that the resulting images are as closely related to the original ones as possible.

Here we can still use the heat equation, but it would be necessary to generate an independent value for entry on the latent space since it includes an UNet as part of the architecture.

Examples of generated images on [3]

Stable diffusion

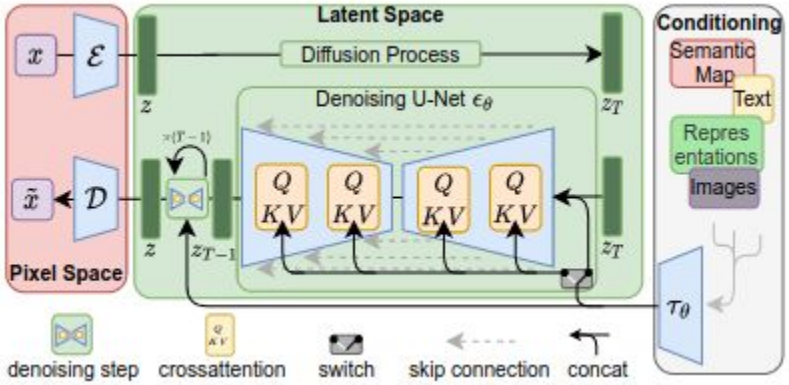
One last model I took a look at and think that may be worthy trying is the Stable Diffusion [5] which is explained quite well in [6]. As you know, Stable Diffusion generates images using texts, but the text part can be easily removed, leaving us with an UNet for the diffusion and using an autoencoder decoder to generate the images.

By using an autoencoder we are no longer restricted to the Fourier transform and allows us to have smaller networks. It's worth noting that we don't have an actual restrain not to use the heat equation for this diffusion process, as long as we assign a different value to each entry on the latent space (to which we get through the autoencoder encoder).

Stable diffusion

One important claim they're making is that they don't need to train the network for different amounts of noise if they use that as part of input. Below you can find the diagram of the architecture in which I would ignore the *semantic conditioning.*

On the next slide, you can find an example of the diffusion process.



Stable diffusion

I could also make the necessary modifications to the networks I have in order to have the amount of noise as an input and use it to determine how many iterations it needs to compute.

References

[1] https://keras.io/examples/vision/siamese_network/

[2] https://arxiv.org/pdf/2006.11239.pdf

[3] https://keras.io/examples/generative/ddim/

[4] https://arxiv.org/pdf/2011.13456.pdf

[5] https://arxiv.org/pdf/2112.10752.pdf

[6] https://jalammar.github.io/illustrated-stable-diffusion/