Image Processing - Exercise 5

Ido Pinto 206483620

1. Image Alignment

before

after

2. GAN Inversion

2.1. Optimization Process

nam_steps=1000, latent_dist_reg_weight=0.1

Original

Initial

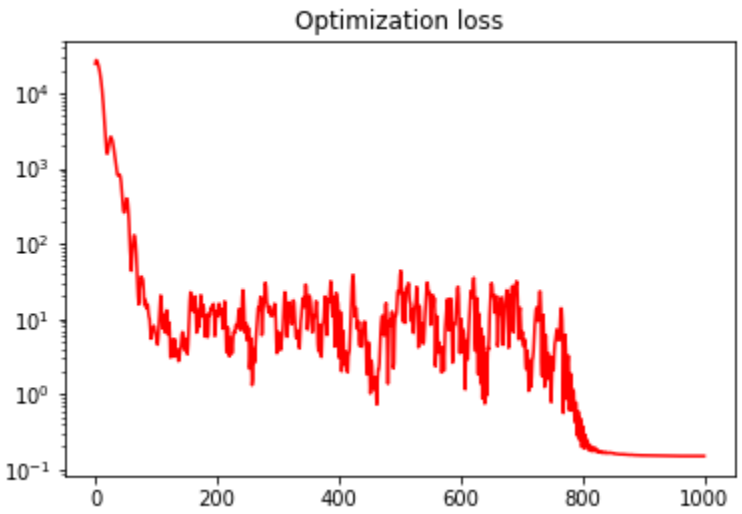
After 100 steps

After 200 steps

After 300 steps

Reconstructed

2.2. Optimization Loss



2.3. Discussion

Latent_dist_reg_weight

The equation for the total loss function is the perceptual loss between the target image features and synthesized image features plus two regularization penalties. One of them is the distance of the latent vector from the mean of all latent vectors in the latent space.

A higher value implies that the loss is calculated based on how close the synthesized image is to the average natural image.

Optimizing this loss will lead to a synthesized image that is more similar to a natural image and less similar to the image being reconstructed, particularly in finer details.

In figure 2.1 I set the weight to 0.1 as my objective is to produce an image that is as similar as possible, and a higher weight would result in less details and more smoothness in the generated image.

Num_steps

The number of steps affects two aspects:

1. It determines the number of iterations in the optimization process of the reconstruction.
   High number of steps is more expensive computationally but will probably converge to a good result.
   Conversely,a low number of steps is less likely to converge and might lead to bad results.
   As shown in figure 2.2, the loss converged after approximately 800 steps, meaning that the process could've been stopped sooner.

2. The learning rate is gradually decreased over the course of the optimization process and is proportional to the num_steps parameter. Choosing the right value will help to ensure that the model has a chance to converge to a good solution before the learning rate becomes too small to make further improvements.

3. Image Reconstruction Tasks

3.1. Image Deblurring

3.1.1. Results

Yann Lecun

The Rock

3.1.2. Discussion

In my solution, if the input was not degraded image, I made a degraded version of it by applying convolving with gaussian kernel and sent the degraded image to the Optimization_steps function.
I made also a degraded version of the synthesized image G generated and used both to calculate the perceptual loss between them, so while G generated good images, by trying to minimize the loss between the two degraded target and degraded synthesized image, G over time will optimally generate a good deblurred image as needed.
If the input was a degraded image (like Yann LeCun) I didn't make a degraded version because it is already degraded, and the process detailed above will do the work.

3.1.3. Issues

3.1.3.1. Implementation Issues

I encountered some implementation issues, one of them was understanding that my input can be either degraded or not degraded and I had to find a solution that handles both cases.
Another issue was that I needed to implement a function in pytorch that builds a gaussian kernel and convolves it with input image. Since I'm not as familiar with pytorch as numpy, I tried to do the kernel build in numpy and convert it to tensor. I also had difficulty understanding how conv2d works and its parameters.

3.1.3.2. Efficiency issues

I noticed that due to the kernel size the runtime of the algorithm increases
My solution after thought was to calculate the kernel tensor just once in the invert_image function and pass it to the latent_optimization so it never needs to be rebuilt.

3.1.4. Hyper-parameters effect

Kernel size

I noticed that too small kernel size can cause blurry result image.
After multiple attempts, size 101 was the best in terms of reconstruction and runtime.
For example this is the generated image with kernel size of 21:

As you can see it is much blurred than with kernel size of 101

latent_dist_reg_weight

I noticed also that low latent_dist_reg_weight is needed to best results
When it wasn't small enough the image was less like the target.
For example this is the result where latent_dist_reg_weight=1

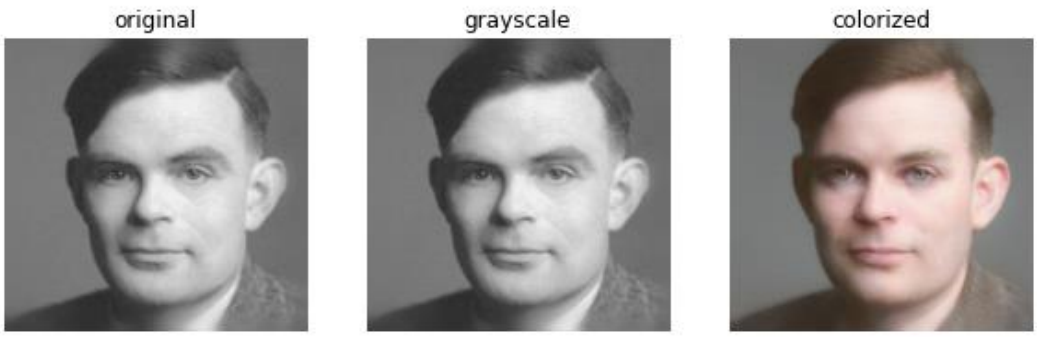
deblurred

The deblurring worked well, but the reconstruction is not faithful to the target image enough.

3.2. Image Colorization

Alan Turing

nam_steps=1000, latent_dist_reg_weight=0.8



The Rock

nam_steps=1000, latent_dist_reg_weight=0.4

3.2.1. Discussion

My solution was to convert the original image (if not degraded) to grayscale by implementing the rgb2gray function in pytorch.
And pass it to the optimization process as the new target. In every step, I converted the generated image by G to grayscale and computed the loss function between them. So over time G will generate images in RGB that in grayscale are very similar to the target image.
Also when choosing the hyper parameters I noticed that latent_dist_reg_weight regularization factor should be high so the generated image colors will be similar to regular human colors (not green) But not too high so the image remain similar as possible to the target image details (rather then colors)

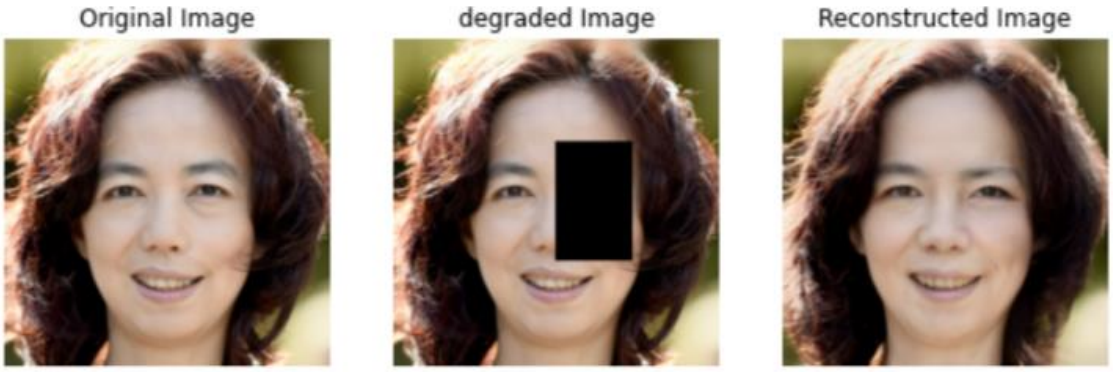
3.2.2. Issues

I ran into a few issues, first I tried to understand how to implement rgb2gray in pytorch. I solved it when I understood that I can exploit tensor properties that the channels are the first dimensions and multiply each channel R, G or B with the corresponding coefficient. (0.299,0.587,0.144)

3.3. Image Inpainting

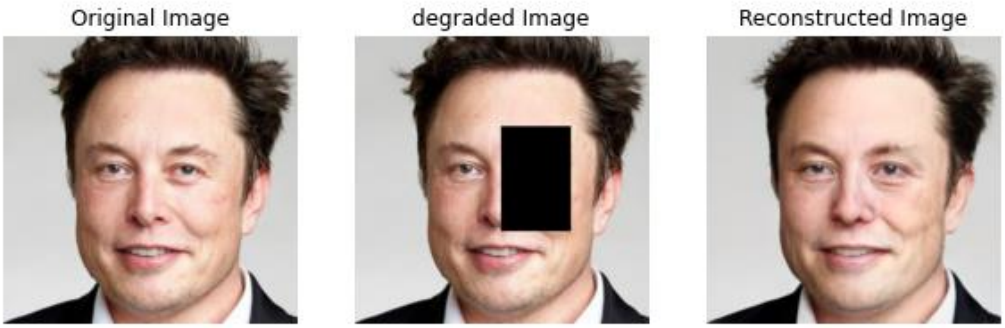
Fei-Fei

num_steps=1000, latent_dist_reg_weight=0.1



Eilon Mask

num_steps=1000, latent_dist_reg_weight=0.1



3.3.1. Discussion

My solution was that given the input image and mask, I applied the mask to the input image by multiplying the binary mask by the image, resulting in the degraded image above.

I passed the mask to the optimization process and applied the mask in every iteration to the generated image by G and calculated the loss between them.

The idea was to force G to create an Image with no degradation that with the mask will be the closest to the target image.

3.3.2. Issues

I encountered a few issues with the implementation.
First, when I just multiplied the mask with the target image I got some errors.
The solution was to divide the mask first by 255 and normalize it
Second, The mask in the generated degraded image was gray and not black.
The solution I found is to replace all the zeros with -1. It fixed the problem.