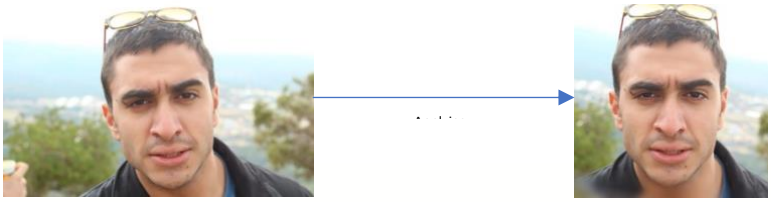
Image Processing – Ex 5

Submitted by Tzlil Ovadia, 311317689.

1. StyleGAN Inversion and Reconstruction

Image alignment

Before we sent any image to be reconstructed by StyleGAN, we needed to perform an alignment first. Here is an example for an alignment done on one of the images I used for this exercise:



On this section, I Applied the `invert_image` function on the image set I presented earlier. I used this function using different values for *`num_steps`* and *`latent_dist_reg_weight`* in order to see how modifications regarding these values affects the resulting image.

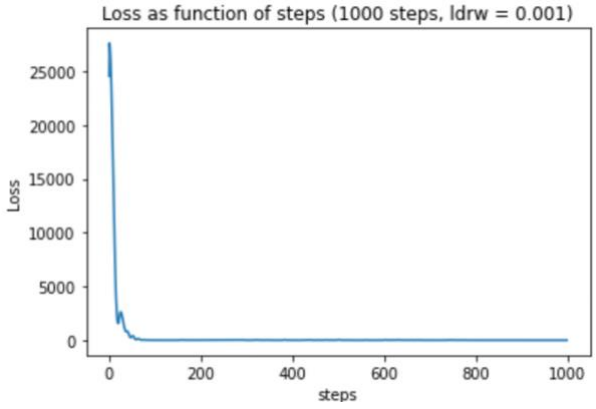
At the beginning, I decided to call `invert_image` function, usin`g the next values for the hyper-parameters; *`num_steps=1000, latent_dist_reg_weight=0.001.`*

And for the next input image:

image.1.1: original image after alignment

This resulted with the next output:

With the next loss function output:



A simple analysis of the loss function shows that the loss value is starting to converge quickly and after only 200 steps, we get from `27643.0078` to `4.8522`, and finally drops to 0.14. In addition, we can see that after reaching step #400, the changes stating to get less significant as we progress, which is somewhat correlated with the loss function.

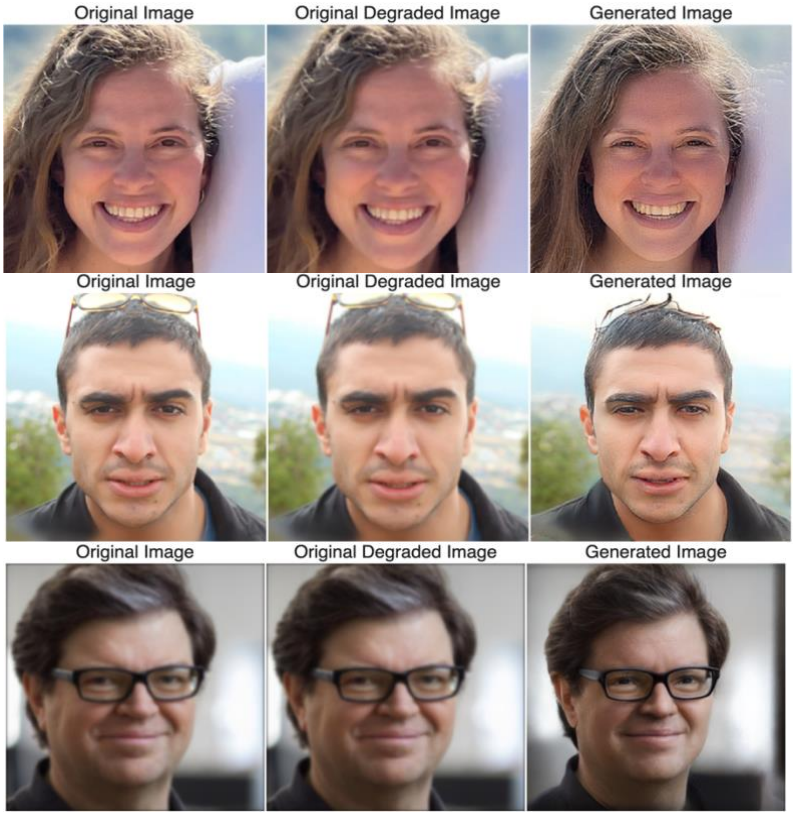
We can conclude that not only that higher value of latent_dist_reg causes the result look different, it is also caused the loss function to take longer until convergence.

2. Image Deblurring

For this section, I implemented a blurring function for a RGB images. The process was as follows:

a) Take the input image and align it.
b) Call `blur_mode` inside `invert_image`, and send the blurred image as the "original" input image to `run_latent_optimization`.
c) Inside the last function, after each iteration the NN outputs a synthetic output image. In order to make the NN result in a deblurred image, we will take the synthetized image, and blur it using the `blur_mode` **before** calculating the loss function.

Reconstruction results



Results analysis

The resulted images shown above were generated using:

- Num_steps = 700, latenet_dist_reg=0.2
- Num_steps = 1000, latenet_dist_reg=0.2
- Num_steps = 700, latenet_dist_reg=0.5

For this section, I implemented a blurring function, using a gaussian 1D vector as a kernel to convolved the images for more efficient calculation (as we learned in class).

Inversion process

The two input images demanded a degradation before sending it to the GAN, while the blurred image of Yann Lecun was already blurred, and needed no prior degradation. I solved it by adding a Boolean parameter to the invert_image function that used as a flag for an already degraded input (used for all the others degradation modes).

Reconstruction process

In the process of reconstructing the images, we needed to manipulate the algorithm, in a way that we intentionally created a deviation from its synthetic result, by degrading the reconstructed image in the same manner of the corruption of the input image. This way, the GAN thinks he generated this corruption and will try to reduce it from step to step.

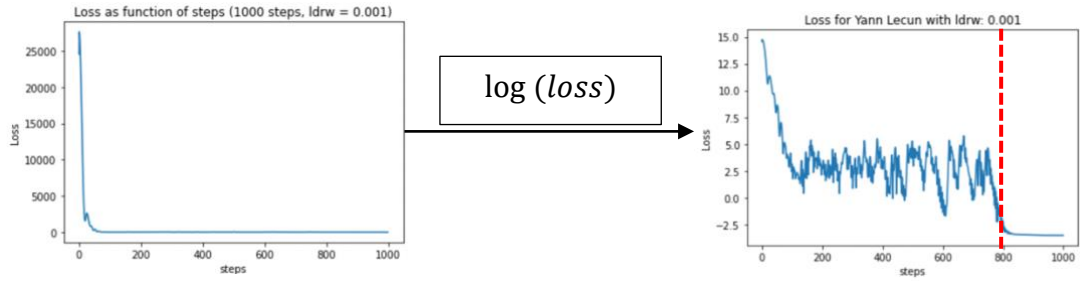
What we did essentially is to take the synth_image, and use the blur_degradation on it before calculation the error. For the 2 input images provided by myself it was pretty much trivial solution due to the fact that any kernel size I'll use, it'd be easy to deblur as long as I kept kernel size constant while the process went on. Yann's case was more complex – We didn't knew what was the kernel size used in order to blur the image, so it took a while to solve this mystery.

Yann Lecun Reconstruction

First, I tried to use different kernel size on the images I provided in order to reproduce the same intensity of the blurring effect in Yann Lecun image. After a little recalibration game, I found that a kernel size n that satisfies $n \in [70,110]$ raises pretty good results. Finally, I tried several values of n in the given range, and settled on the value of 100 which raised the resulted image above.

In addition to the kernel size, I tried playing with the hyper parameters in order to get better results.

Num_steps - In every optimization process, I plotted the loss function, but instead of plotting it as is, I used a logarithmic scale in order to find the "Sweet point" where the NN is really converged. As we can see in the first section, where we provided the so-called regular loss function, it is really hard to tell where the graph was really converged, and where it wasn't.



By using the Log transform, we get much better separation, and we could see details we couldn't do without. So, I chose to use the num_steps according to where it converges on the graph somewhere near the red line, with a safety buffer, given a latent_dist_reg value. Further then that would be redundant.

latent_dist_reg – By changing the value of this parameter, we change the manner the reconstructed image would be "distant" from the real image. On one hand, a small value (0.001) gave us a good deblurring effect, but on the other hand we could see some imperfections which looked really noisy when image was fully scaled. So, I tried tweaking the values a little, and found that around 0.5 we get an image that both deblurred an smooth. My assumption is that by constraining the GAN to a low value for this parameter, can be a little problem for situations when we don't know what the source cause of the corruption is, i.e. the kernel used to deblur the image. So, by increasing the value of this parameter, we give the GAN some level of freedom to make it look more realistic image.

Latent_dist_reg=0.001

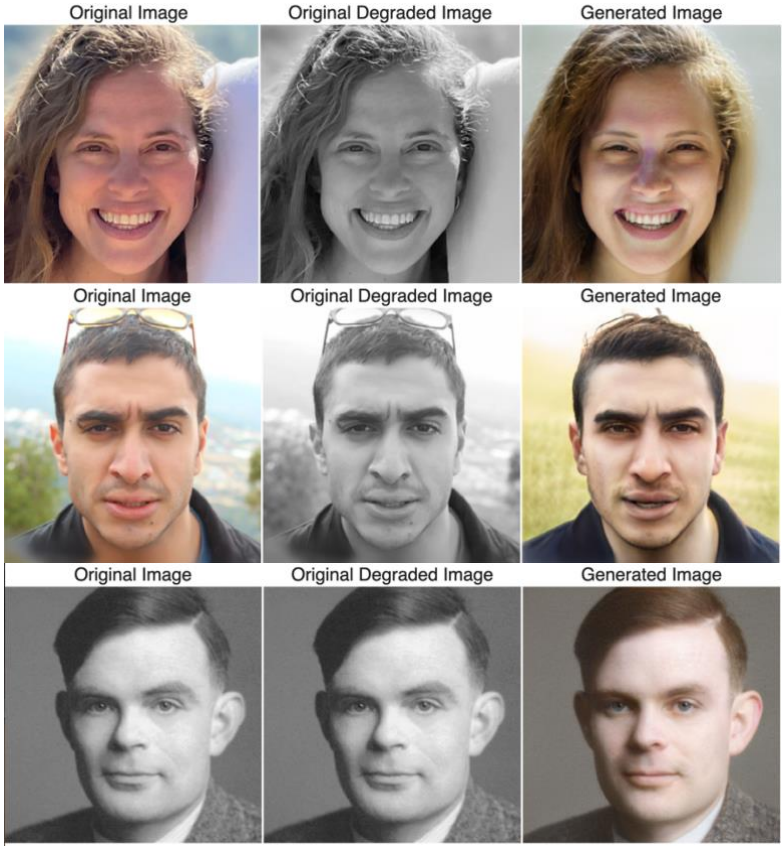
Latent_dist_reg=0.5

3. Image Colorization

For this section, I implemented a basic function which returns a greyscale image, given RGB images. The process was as follows:

a) Take the input image and align it.
b) If the input image is not already corrupted, we'll call `colorization_mode` inside `invert_image`, and send the greyscale image as the "original" input image to `run_latent_optimization`.
c) Inside the last function, after each iteration the NN outputs a synthetic output image. To make the NN deliver a colorized reconstruction of the image, we will take the synthetized image the GAN is generated on each step and convert it to a greyscale image using the `colorization_mode` **before** calculating the loss function.

Reconstruction results



Result analysis

The resulted images shown above were generated using:

- Num_steps = 900, latenet_dist_reg=0.6
- Num_steps = 900, latenet_dist_reg=0.3
- Num_steps = 500, latenet_dist_reg=0.5

Inversion process

Just as the previous section, the two input images demanded a degradation before sending it to the GAN, while Alan's was already a greyscale image, and needed no prior degradation.

Reconstruction process

What we did essentially is to take the synth_image variable in the run_latent_optimization function, and applied the greyscale_mode on it before calculation the error.

As we can see, both the provided image of Alan Turing, and the Images provided by myself, were generated using latenet_dist_reg's values which are high. After iterating over some values, starting from 0.001 up to almost 0.8, I concluded that using too small value brings an odd-looking image. On one hand, the generated image was loyal to the facial structure, but on the other hand, the colors were unnatural and not coherent. On the other hand of the scale, choosing a higher value for latenet_dist_reg's, brought a natural and coherent colorization, BUT the facial structure wasn't loyal to the original image.
Let us see some poor reconstructions which brought after choosing extremely low/high values of the latenet_dist_reg's parameter:

Finally, after playing with the values again and again, I found the "sweet spot" for each of the images as I mentioned earlier.

4. Image Inpainting

For this section, I implemented a function used to mask the image in a manner which mimics the making we asked to perform. The way I implemented it was as follows:

- Estimate what are the boundaries of the original masking provided for this exercise.
- Set all the pixels which falls inside the boundaries to zero.

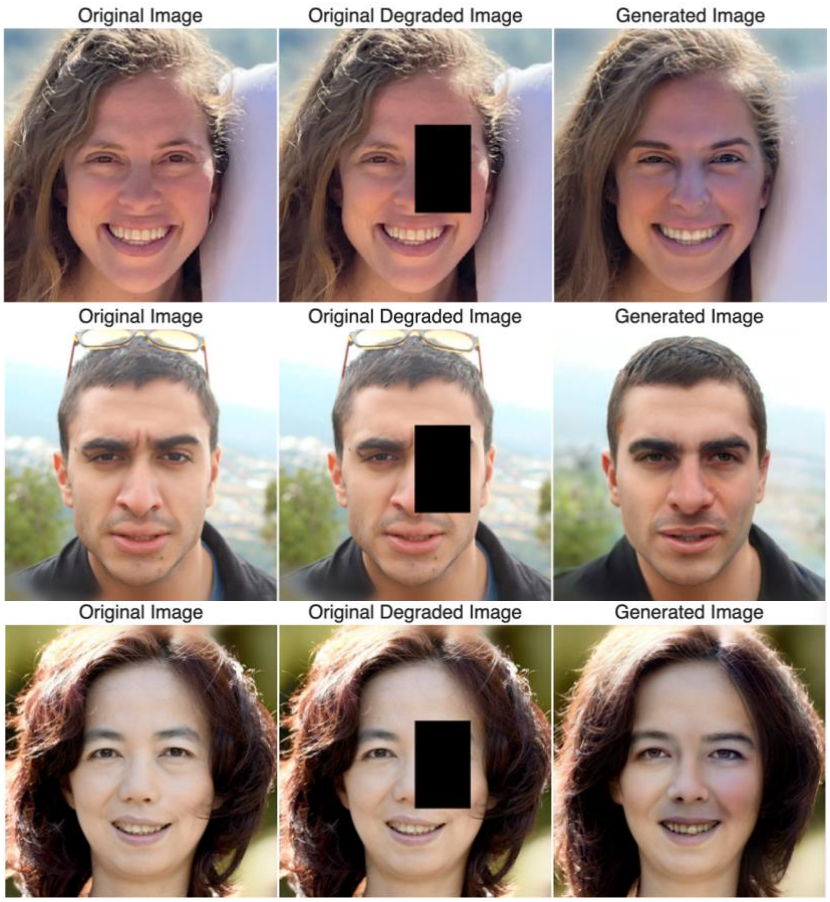
From here, all we got left to do is to figure out is where we should call the degrading function inside run_latent_optimization such that the image reconstruction raises a good result.

The first try was using the naïve approach I took on previous tasks, and it was calling it inside the dedicated section as described inside the Colab notebook. The results brought using this approach were odd looking, and, while watching the process as it goes, I noticed that the mask becomes grey, and realized it happened due to the performing down sampling to the synthesized image, which also changes the results.

Another approach I took, was trying to manipulate the mask image bring a black mask, but this approach, again, brought some troubling looking results.

Finally, after some time playing with the code, I figured out that maybe the thing that I needed to change was the point in the code where I call the degrading function, instead of changing the degrading function itself. I figured out that the location which brings the wanted results was after the down sampling, before calculating the loss.

Reconstruction results



Result analysis

The resulted images shown above were generated using:

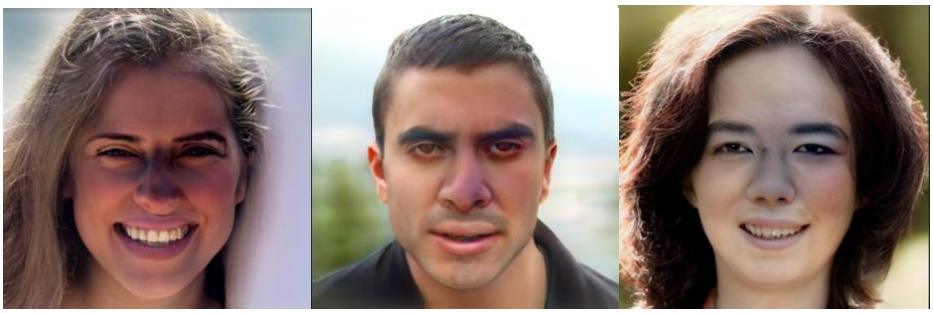
- Num_steps = 1000, latent_dist_reg = 0.4
- Num_steps = 1000, latent_dist_reg = 0.5
- Num_steps = 750, latent_dist_reg = 0.001

Reconstruction process

Along the process, I have dealt with a lot of problems. Starting with the color of the mask, through choosing bad hyper-parameters and resulting with weird images.

For some reason, this reconstruction was more sensitive to the hyper-parameters, both number of steps and latent_dist_reg. I've noticed that some combinations with too large steps number brought a bad coloring effect on the previously masked zone, so I tried not going too far with the steps in order to keep colors more natural. Another thing is

Here is a summarization for some of the poorly resulted trials:



Conclusions

In all of the corrupted images to be reconstructed, the workplan was similar – applying the same degradation as the input image provided from invert_image on the synthesized images that been generated on each NN iteration, in order to manipulate the NN to 'force' it to think that the corruption is made by it, because this manipulation causes the loss calculation to be affected by this degradation – and next time, the NN would try to fix it.

I saw how the hyper-parameters acts on the network, and how we can affect the result of the NN. The two parameters are sort of acting together – incrementing the weights should come with higher steps until convergence.

Another interesting thing that I learned is when I used StyleGAN in order to reconstruct an image from a corrupted image – by doing so, I came to realize that we can modify the weights and get sometimes better results for higher values, but there is a tradeoff which translates by some abnormalities, e.g., unnatural colors, uncoherent images etc.

After playing with the code for days and nights, I learned a lot about Deep learning, but simultaneously saw how complicated it can be. There is a lot for me to learn, and I was really happy to work on this exercise!

I really think that there is a lot to accomplish with this exercise, and I think that it could be very helpful if next semester, the staff would offer to the student further explanation regarding the Pytorch modules.

Thank You!