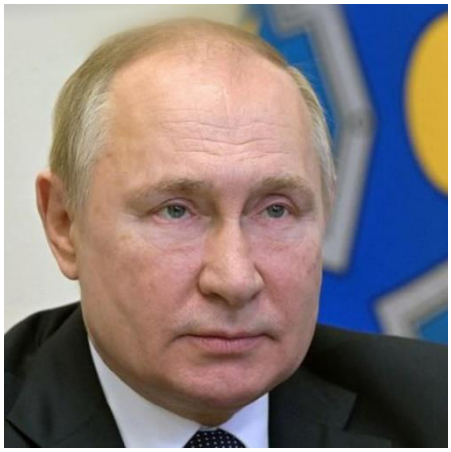
Ex5 image processing

Amitai Ovadia 312244254

3.1 Image Alignment:

In order to use the GAN network the images had I fed to the net had to undergo an alignment to the standard GAN train samples. For this task I used the supplied align_faces.py, and here is an example.

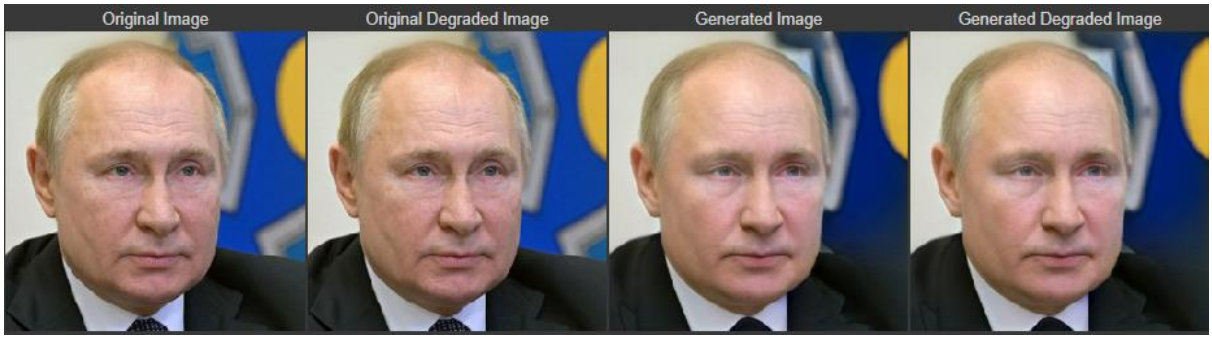


3.2 GAN inversion:

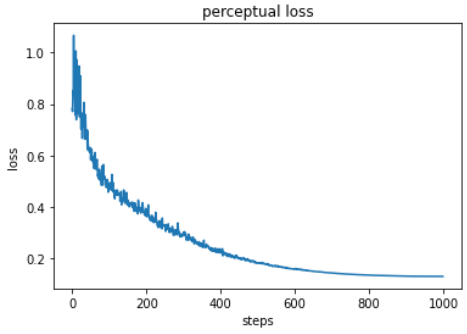
In this task we were asked to run the optimization process on an image of our choosing and "play" with the different hyper parameter of the net: number of iterations and the latent distance.

I have compared the different results by my subjective opinion, and also by the loss function's values in the end of the run.

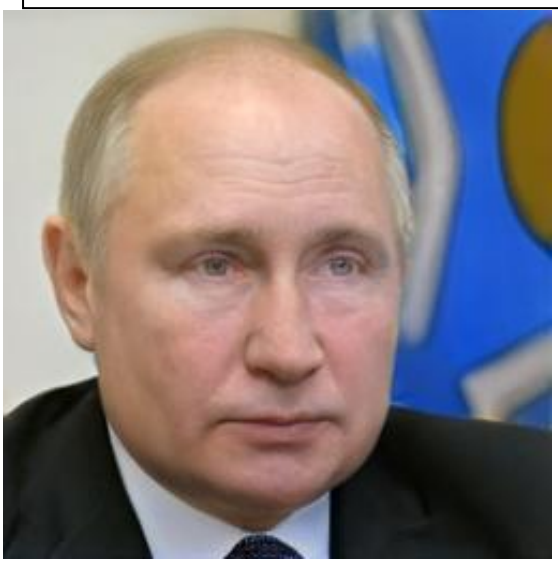
1000 iterations, latent distance = 0.001, perceptual loss: 0.13 , loss: 0.13

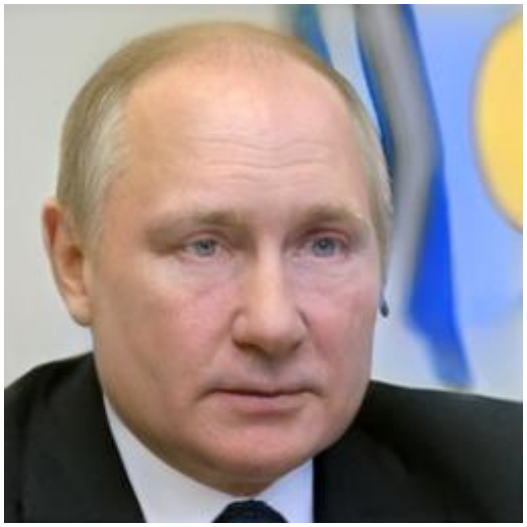


in the process I have also plotted the perceptual loss as a function of the number of iterations: it's clear that the loss goes down nicely and converges after ~900 iterations



1000 iterations, latent distance = 0.1, perceptual loss: 0.13 , loss: 0.17

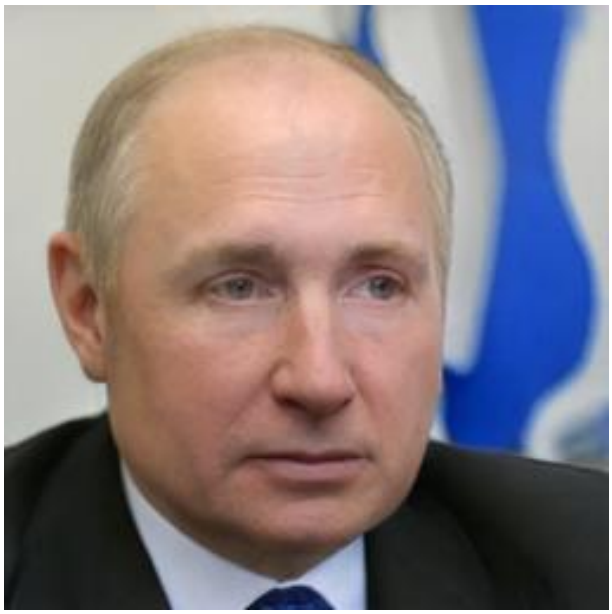




(on the left) **best results**

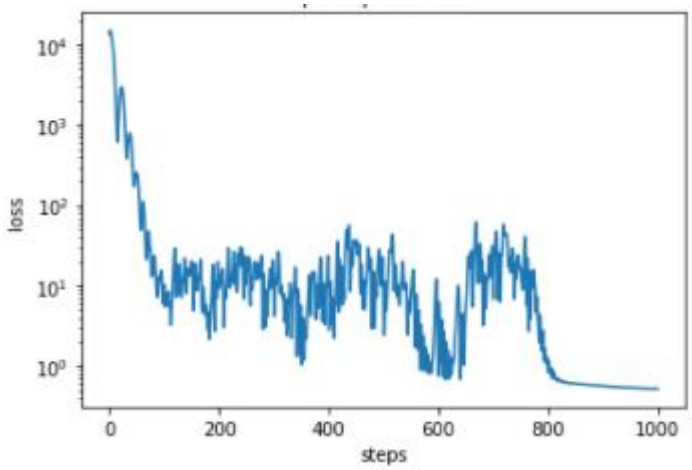
1000 iterations, latent distance = 0.2, perceptual loss: 0.10 , loss: 0.15

1000 iterations, latent distance = 1, perceptual loss: 0.21 , loss: 0.28 (on the right)

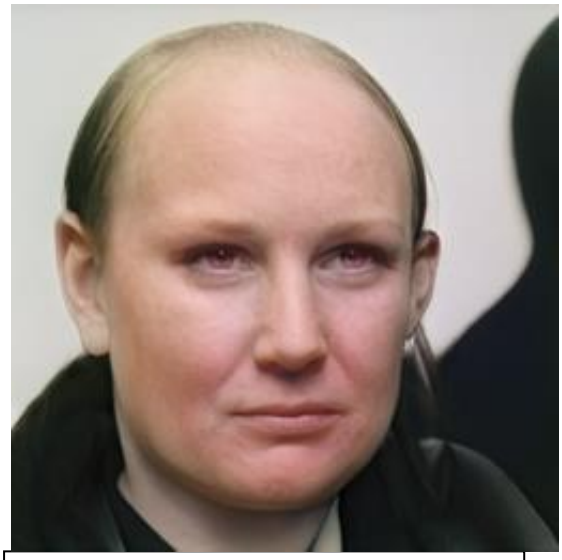


1000 iterations, latent distance = 10, perceptual loss: 0.47 , loss: 0.52

i's clear from the generated image and from comparing the different loss values that we got the best Putin by choosing a latent distance of 0.2. on the right there is a plot of the lost function of the last run in log scale. We can see in the graph that the loss function converges around ~800 iterations.

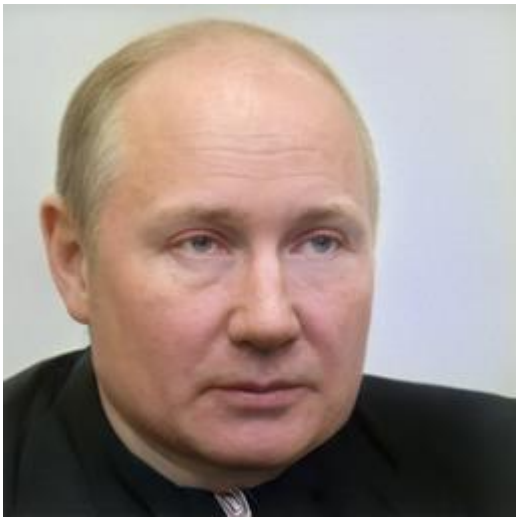


Now I tried to play with the number of iterations:



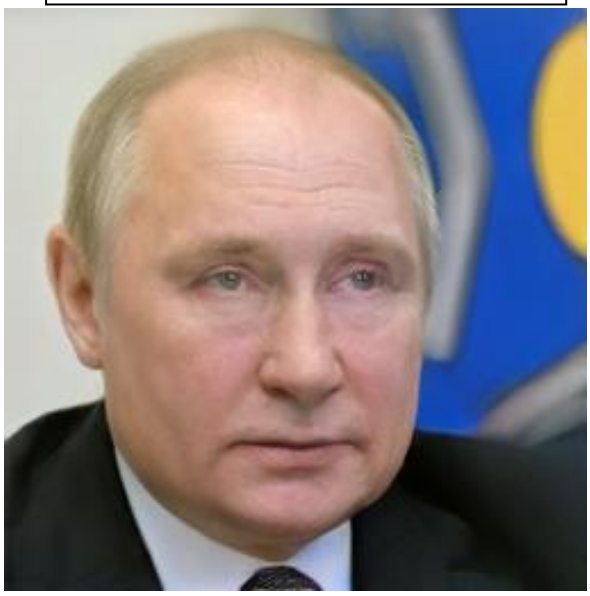
60 iterations, latent distance = 0.2, perceptual loss: 0.36 , loss: 27.48:

It's clear that the loss is still a 100 times bigger and the Image is still not reconstructed proparly

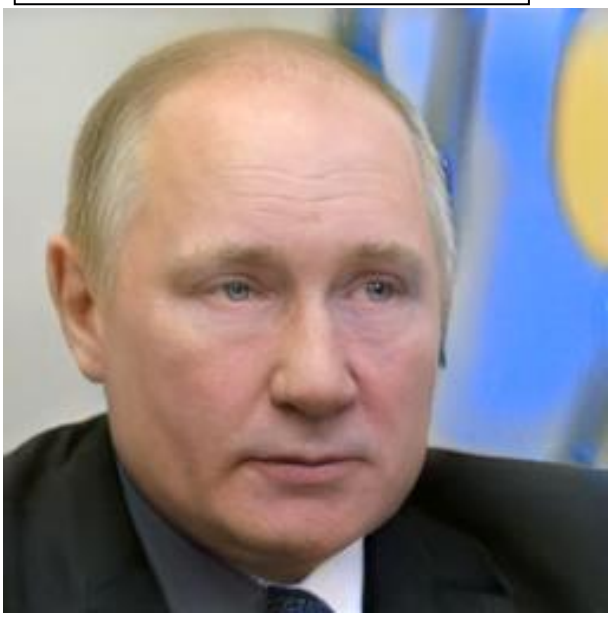


200 iterations, latent distance = 0.2, perceptual loss: 0.28 , loss: 0.34:

Much better then the 10 distance but no so remote from the 1



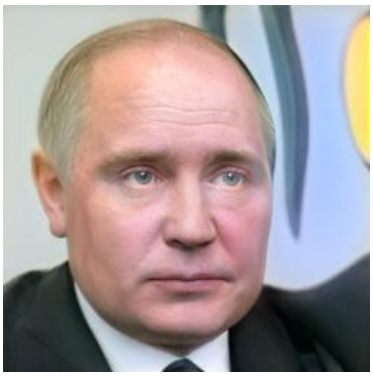
3000 iterations, latent distance = 0.2, perceptual loss: 0.09 , loss: 0.14



800 iterations, latent distance = 0.2, perceptual loss: 0.15 , loss: 0.2

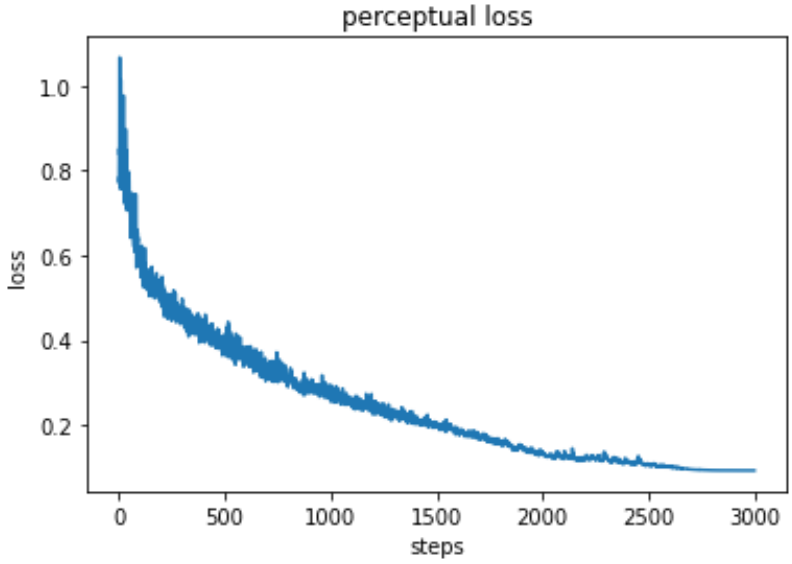
We can see that in comparison to the 1000 iterations it improves only a little: the perceptual loss goes form 0.15 to 0.1 and I couldn't see no visual difference

A weird phenomenon I noticed was that in the 3000 iterations run, the network converged much slower: here is the generated image after the first 1000 iterations (on the left): it's clearly worse then the run with "only" 1000 iterations:



Furthermore, here is a graph of the perceptual loss vs number of iterations: It's also visible that it converges in much a smaller rate then the 1000 iterations graph above.

My theses is that the network chooses it's learning rate as a function of the number of iterations: smaller learning rate for a high number of iterations and vise versa: so that the perceptual loss looks very similar every time.



Another example: original (on the left) vs generated.

1000 iterations, latent distance = 0.2, perceptual loss: 0.16 , loss: 0.22

3.3.1 Image Deblurring

In this section, we tried to reconstruct a blurred image, by blurring the generated image before it's compared to the original image and the loss is calculated. This way the network's goal is to generate an image that after blurring, will be as close as possible to the blurred input image.

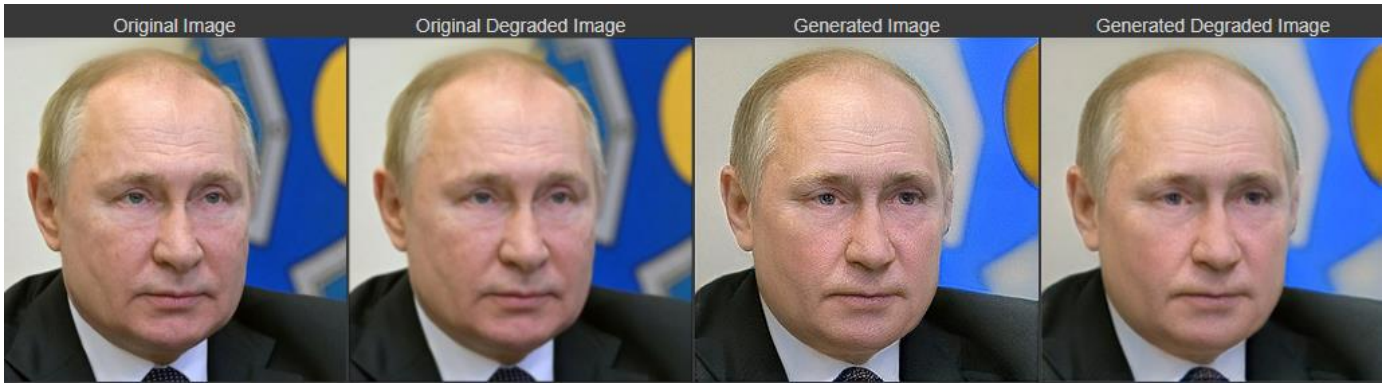
In this section I have implied a convolution with a 2d gaussian kernel of different sizes, the 1d kernel of size n was calculated using the n first binomial coefficients, and then by convoluting itself with it's own transpose was the 2d gaussian kernel generated.

The kernels were applied first on the original image as an input to the network, and second on the generated image so the loss will be computed between the 2 blurred images.

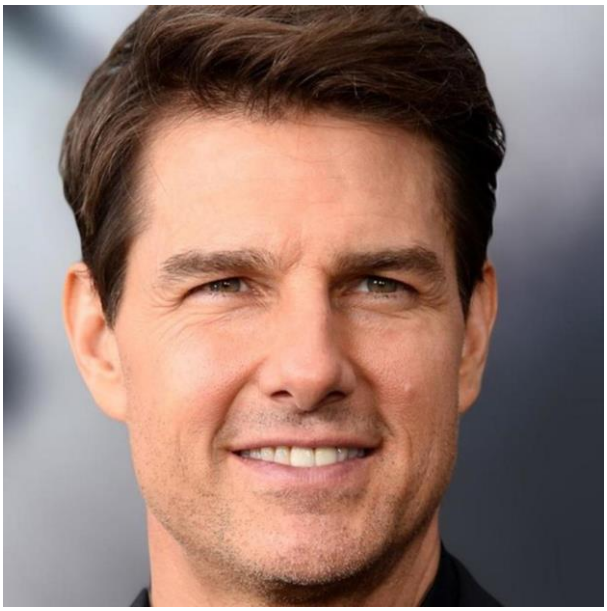
I have tried several kernel sizes on

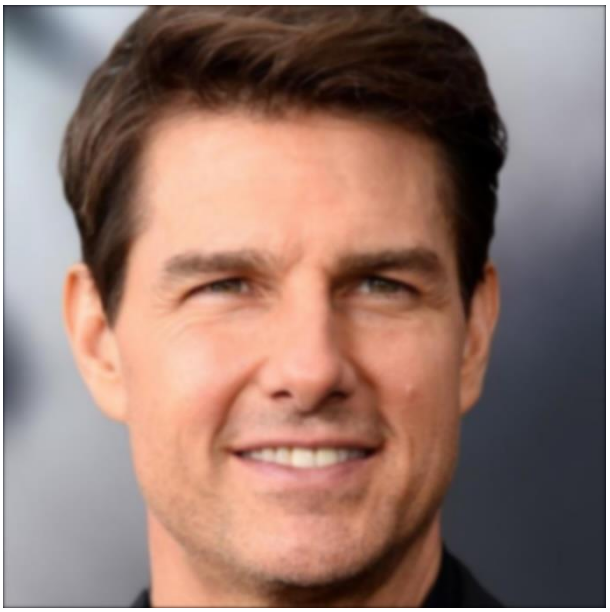
1000 iterations, latent distance = 0.2, perceptual loss: 0.13 , loss: 0.18

Kernel size = 35



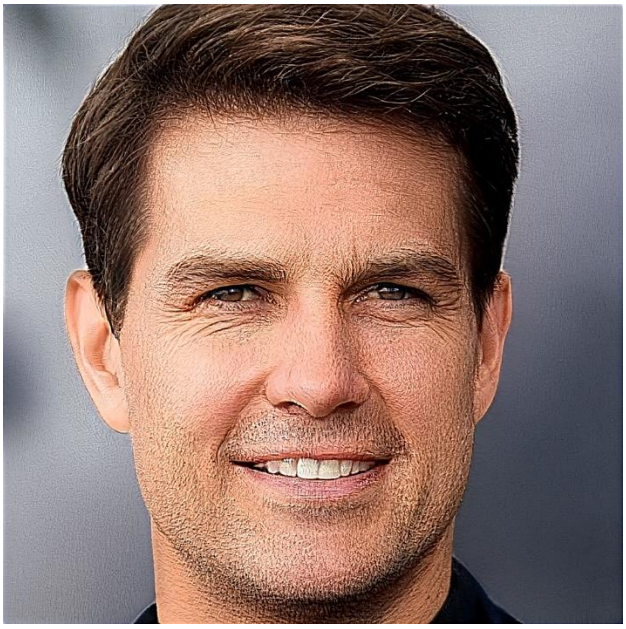
1000 iterations, latent distance = 0.2 Kernel size = 35

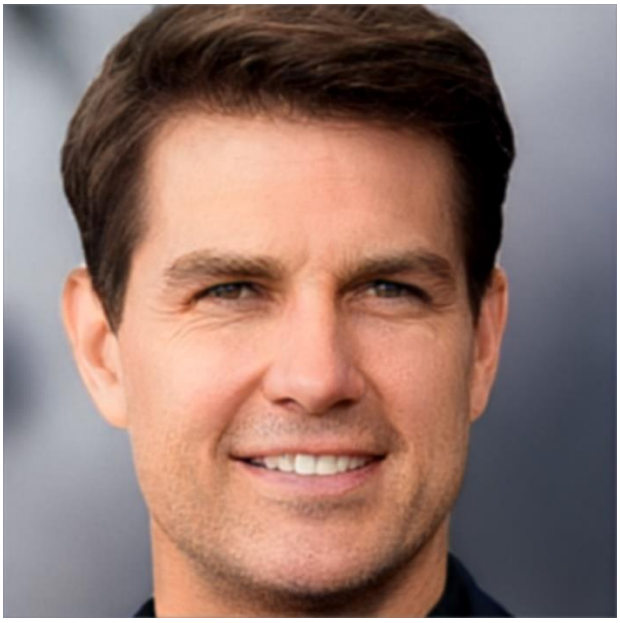




Generated image: after blurring looks very close to the blurred original image (on the right)

But, the generated image without the blurring looks too detailed.



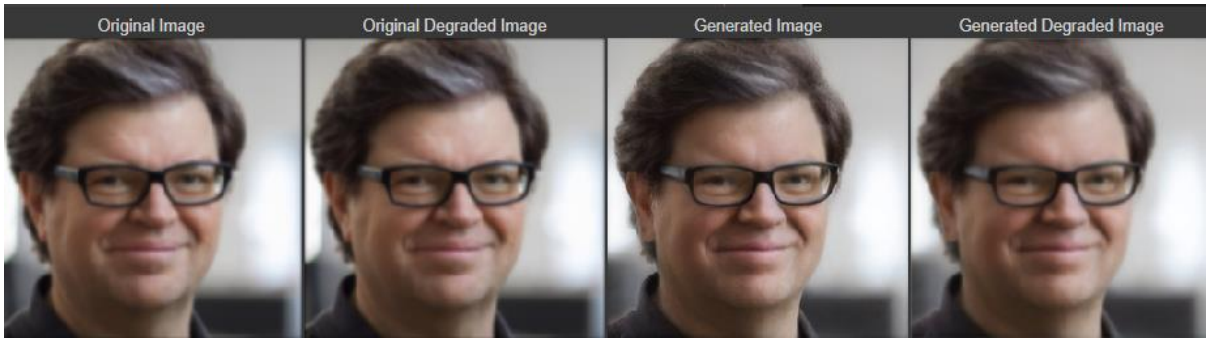


Next, I tried to deblur the supplied **Yann LeCun image**. For this run I didn't blur the input image because it's already blurred, and I tried different kernels, but the only one that worked a little bit was 50:

Kernel size = 50 percp_loss = 0.08, loss = 0.13

It's not a very good result. Probably it's because It's not the right kernel size that takes the original unblurred image to it's current blurred version (the input to the nework). That's why the blurred version of the generated image didn't look like the blurred Yan image and the optimization process didn't lead to regeneration of a good unblurred image.

Kernel size = 50 percp_loss = 0.07, loss = 0.11 **best result closer to the original blurring**



3.3.2 Image Colorization

In this task I tried to recolor grayscale images. The idea, similar to the use of the blurred one, was to input the grayscale image into the network, and turn the generated image into a grayscale image before the comparison is made so the loss is computed by comparing the original grayscale to the generated image after turning It to grayscale.

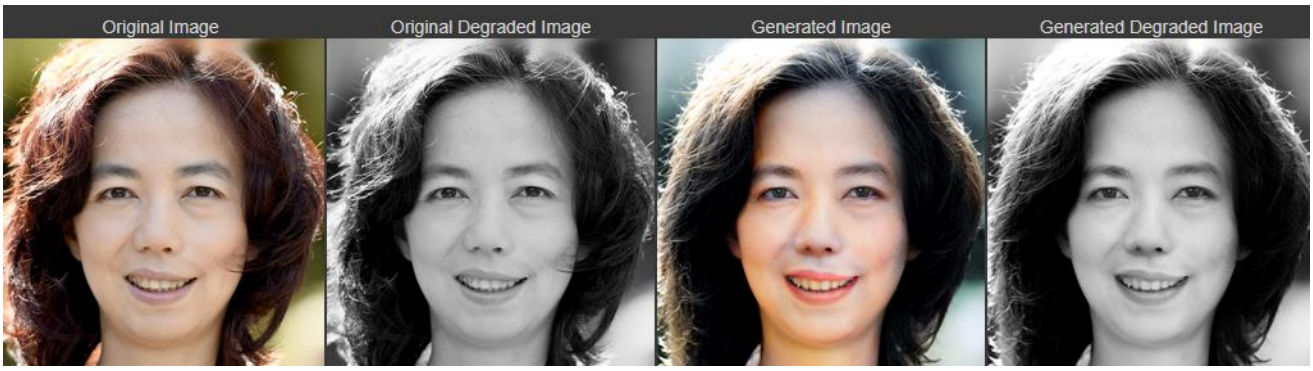
The 'graying' is done by Torchvision.transforms.Grayscale function and outputted 3 channels of colors so that the feature comparison could take place.

The only issue I encountered was the latent distance, the Turing image reconstruction was a bit different with different latent distances.

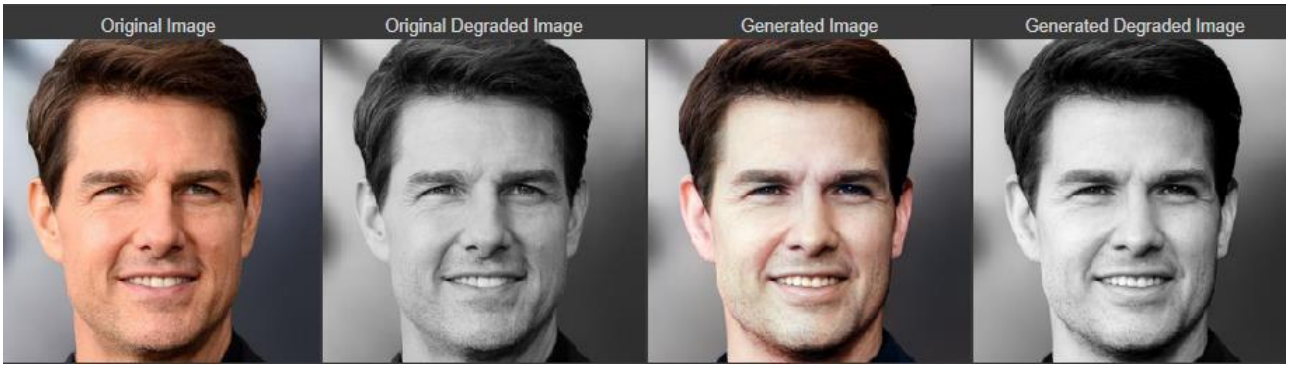
Latent distance = 1, Prep_loss = 0.16, loss = 0.21 (the middle one) closer to the ex5 pdf example

Latent distance = 0.2, Prep_loss = 0.12, loss = 0.16 (the right one) **smaller losses**

Latent distance = 0.2, Prep_loss = 0.28, loss = 0.32



Latent distance = 0.2, Prep_loss = 0.23, loss = 0.26

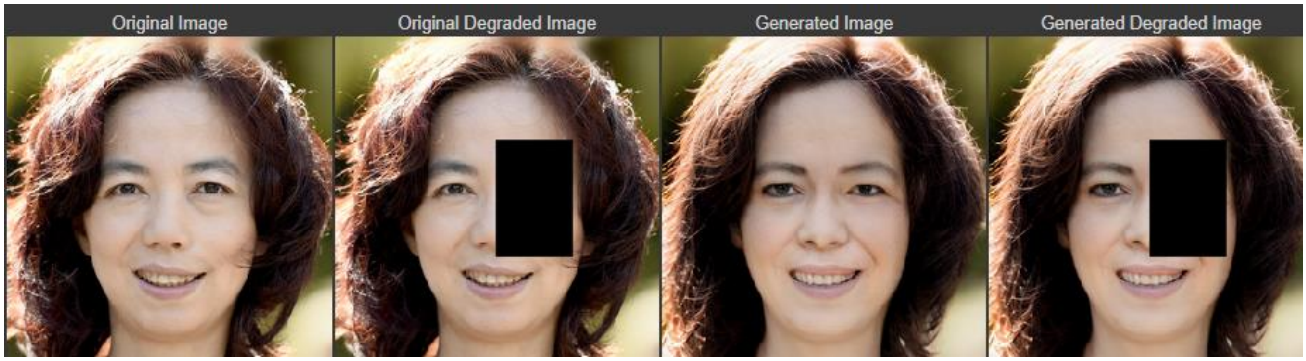


3.3.3 Image Inpainting

The same trick is done here like previous parts.

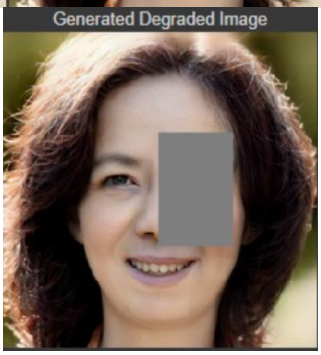
The generated image is being masked like the original image, so the network is trying to generate an image that after applying a mask we look the same as the masked original.

latent size = 0.5, percep_loss = 0.28, loss= 0.33 **best result**

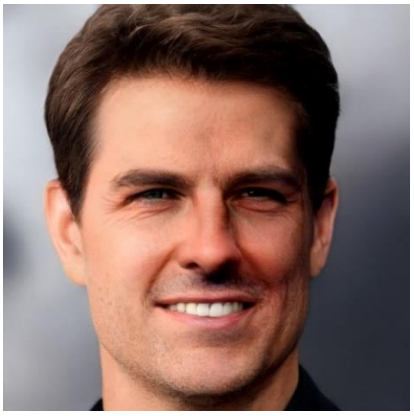


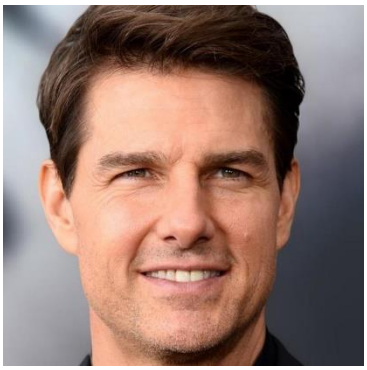
Latent size = 0.2, percep_loss = 0.25, loss= 0.29

In this part I got a gray mask instead of black in the generated image. My solution was to add the mask again and subtract 1.



Tom cruz with latent distance 0.2 (original on the bottom right, generated in the middle), not a very good reconstruction around mask area





latent distance = 0.2, prec_loss = 0.19, loss = 0.22 not a good reconstruction probably because the image wasn’t in a good enough quality/