4xRealWebPhoto_v2

Paired Dataset Preparation Summary

Purpose

Goal: A 4x paired dataset that can be used to train sisr models for upscaling photos downloaded from the web.

Usecase: Person takes photo, uploads it on the web. Another person downloads that image, and re-uploads it on the web. We now download this image and upscale it.

Dataset: Apply degradations to a photo dataset.

Degradations simulating the usecase: Apply some realistic noise (my ludvae model) and realistic blur (weak lens blur) to simulate a photo that might have noise&blur, then scale and compress it (treatment applied by web service provider when uploading), re-scale and re-compress it (again, service provider when uploading).

(To be honest, this is the third iteration of this dataset, previous learnings carried into this one)

Simulating use case (me and a friend in the background goofing off / having fun)

musl 03.11.2022 18:26

Dataset Name: Nomos8k
License: check sources
Download Link (3.4GB):
https://drive.google.com/file/d/1ppTpi1-FQEBp908CxfnbI5Gc9PPMiP3l/view

SHA256: 89724f4adb651e1c17ebee9e4b2526f2513c9b060bc3fe16b317bbe9cd8dd138
Purpose: Realistic SR
Description:

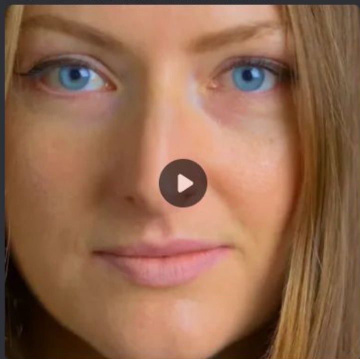
Contains 8507 tiles of 512x512px each, and the `meta_info` file. Nomos8k is meant to be an improvement over the previous. Just like Nomos2k, the purpose of this dataset is to gather as much information about textures and shapes as possible. The original images were tiled to 512px squares and hand selected based on the same criteria used on nomos2k:

- High signal-to-noise ratio (low noise)
- Diverse.
- Sharp (no motion blur, shallow DOF is accepted but undesired)
- Contains mixed and complex textures/shapes that cover most part of the image

Additionally, Nomos8k has been enhanced by around ~2.5k tiles of human textures, and ~400 faces selected from the FFHQ dataset.

Raw images were processed on rawtherapee using prebayer deconvolution, AMaZe and AP1 color space. Sources: Adobe-MIT-5k, RAISE, FFHQ, DIV2K, DIV8k, Flickr2k, Rawsamples, SignatureEdits, Hasselblad raw samples and Unsplash.

Feel free to mirror this dataset. (Bearbeitet)



Input Dataset

datasets

If you don't have a dataset, you can either download research datasets like DIV2K or use one of the following.

- `nomos_uni` (*recommended*): universal dataset containing real photographs and anime images
- `nomos8k`: dataset with real photographs only
- `hfa2k`: anime dataset

These datasets have been tiled and manually curated across multiple sources, including DIV8K, Adobe-MIT 5k, RAISE, FFHQ, etc.

dataset	num images	meta_info	download	sh
nomos_uni	2989 (512x512px)	nomos_uni_metainfo.txt	GDrive (1.3GB)	6403764c3062aa8aa6b84231950200
nomos_uni (lmdb)	2989 (512x512px)	-	GDrive (1.3GB)	596e64ec7a4d5b5a6d44e098b12c2e
nomos_uni (LQ 4x)	2989 (512x512px)	nomos_uni_metainfo.txt	GDrive (92MB)	c467e078d711f818a0148cfb097b3f6
nomos_uni (LQ 4x - lmdb)	2989 (512x512px)	-	GDrive (91MB)	1d770b2c6721c97bd2679db68f43a9
nomos8k	8492 (512x512px)	nomos8k_metainfo.txt	GDrive (3.4GB)	89724f4adb651e1c17ebee9e4b2526
hfa2k	2568 (512x512px)	hfa2k_metainfo.txt	GDrive (3.2GB)	3a3d2293a92fb60507ecd6dfacd636a

Processing 1. Step: Multiscale

I multiscale the dataset to improve model learning for crop size (for example, bush leaves look different from close up and from afar).

The dataset consists of 512x512px images. I use 1, 0.75, 0.5 and 0.25 multiscale so we get 512x512, 384x384, 256x256 and 128x128 px images. This means max gt_size is 128.

Normally higher crop means better quality output for transformer, since the model gets more context. But, since I use multiscale, I expect a similar amount of information for the model to learn from (since the crop of the smallest image contains the full image) while training will be faster (so a trade-off for increased training speed while quality hit should not be bad. Alternatively I could also have gone with 0.5 as lowest scale with therefore 256 as highest crop size - I did that at first, but changed based on above thoughts).

Multiscaling this way also increase the number of images in the dataset from 8492 to 33968, which will also improve degradation distribution, since the degradations are randomized pre-applied, instead of images being on-the-fly randomized degraded during training (so static instead of dynamic degradation during training).

```python
import argparse
import glob
import os
from PIL import Image


def main(args):
    # For nomos8k I consider 0.75, 0.5 and 0.25 as scales, so we end up with 512x512, 384x384, 256x256 and 128x128. Meaning highest gt_size to set for training is 128. I could also have gone with only 0.75, 0.5 scales to end up with smallest 256x256 images so highest gt_size. Normally, higher crop means higher quality for transformer training, but I think that when using multiscale the model might be able to learn a similiar amount of information since the lowest crop includes the full input, while training will be faster.
    scale_list = [1, 0.75, 0.5, 0.25]

    path_list = sorted(glob.glob(os.path.join(args.input, '*')))
    for path in path_list:
        print(path)
        basename = os.path.splitext(os.path.basename(path))[0]

        img = Image.open(path)
        width, height = img.size
        for idx, scale in enumerate(scale_list):
            print(f'\t{scale:.2f}')
            rlt = img.resize((int(width * scale), int(height * scale)), resample=Image.LANCZOS)
            rlt.save(os.path.join(args.output, f'{basename}T{idx}.png'))

if __name__ == '__main__':
    """Generate multi-scale versions for GT images with LANCZOS resampling.
    It is now used for the nomos8k dataset
    """
    parser = argparse.ArgumentParser()
    parser.add_argument('--input', type=str, default='datasets/nomos8k', help='Input folder')
    parser.add_argument('--output', type=str, default='datasets/nomos8k_multiscale', help='Output folder')
    args = parser.parse_args()

    os.makedirs(args.output, exist_ok=True)
    main(args)
```

```
        1.00
        0.75
        0.50
        0.25
gt/3699.png
        1.00
        0.75
        0.50
        0.25
gt/3700.png
        1.00
        0.75
        0.50
        0.25
gt/3701.png
        1.00
        0.75
        0.50
        0.25
gt/3702.png
        1.00
        0.75
        0.50
        0.25
gt/3703.png
        1.00
        0.75
        0.50
        0.25
gt/3704.png
        1.00
        0.75
        0.50
        0.25
gt/3705.png
        1.00
        0.75
        0.50
        0.25
gt/3706.png
        1.00
        0.75
        0.50
        0.25
gt/3707.png
        1.00
        0.75
        0.50
        0.25
gt/3708.png
        1.00
        0.75
        0.50
        0.25
gt/3709.png
        1.00
        0.75
        0.50
        0.25
gt/3710.png
        1.00
        0.75
        0.50
        0.25
gt/3711.png
        1.00
        0.75
        0.50
        0.25
gt/3712.png
        1.00
```

Name:	nomos8k
Type:	Folder (inode/directory)
Contents:	8’492 items, totalling 6.7 GB

Name:	nomos8k_multiscale
Type:	Folder (inode/directory)
Contents:	33’968 items, totalling 7.2 GB

Applying multiscale to nomos8k

Processing Step 2: Blur

In this step, I am applying realistic lens blur with the help of python script i made using NatLee/Blur-Generator from github. Since in my previous experiment the model had a bit of trouble handling realistic lens blur, this time I opted to duplicate each image but with lens blur radius 2 or 3 applied (see appendix), meaning the network will have a non-blurry and blurry version of each image to learn from. My idea is that this should help the trained sisr model handle non-blurry as well as blurry input images better.

Of course this step also increased dataset size because of this duplication approach.

Blur Generator

python 3.6 | 3.7 | 3.8 | 3.9 | 3.10 | 3.11 implementation cpython

Test passing Release passing

status stable license MIT

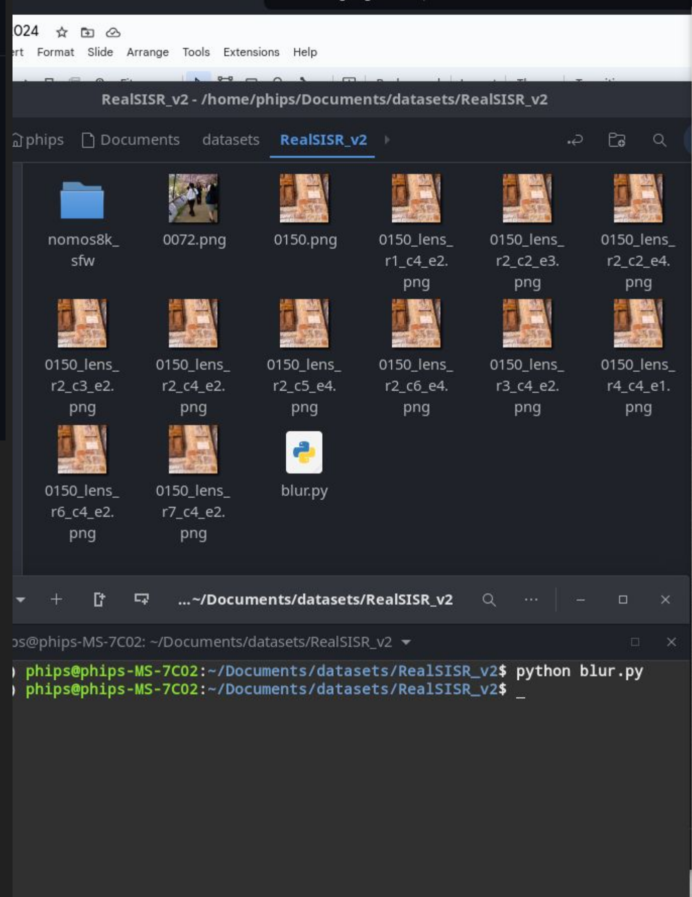
pypi package 1.0.4

downloads 436/month downloads 49/week downloads 9/day

MADE WITH PYTHON

This tool is for generating blur on images.

There are 3 types of blur modes of `motion`, `lens`, or `gaussian`.

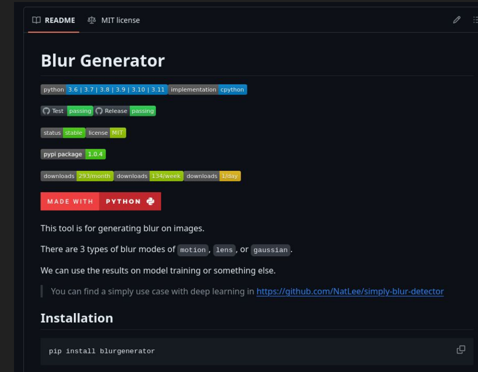



```python
import cv2
from blurgenerator import lens_blur

img = cv2.imread('0150.png')
result = lens_blur(img, radius=1, components=4, exposure_gamma=2)
cv2.imwrite('./0150_lens_r1_c4_e2.png', result)

img = cv2.imread('0150.png')
result = lens_blur(img, radius=2, components=4, exposure_gamma=2)
cv2.imwrite('./0150_lens_r2_c4_e2.png', result)

img = cv2.imread('0150.png')
result = lens_blur(img, radius=3, components=4, exposure_gamma=2)
cv2.imwrite('./0150_lens_r3_c4_e2.png', result)

img = cv2.imread('0150.png')
result = lens_blur(img, radius=4, components=4, exposure_gamma=2)
cv2.imwrite('./0150_lens_r4_c4_e1.png', result)

img = cv2.imread('0150.png')
result = lens_blur(img, radius=5, components=4, exposure_gamma=2)
cv2.imwrite('./0150_lens_r5_c4_e2.png', result)

img = cv2.imread('0150.png')
result = lens_blur(img, radius=6, components=4, exposure_gamma=2)
cv2.imwrite('./0150_lens_r6_c4_e2.png', result)

img = cv2.imread('0150.png')
result = lens_blur(img, radius=7, components=4, exposure_gamma=2)
cv2.imwrite('./0150_lens_r7_c4_e2.png', result)

img = cv2.imread('0150.png')
result = lens_blur(img, radius=2, components=3, exposure_gamma=2)
cv2.imwrite('./0150_lens_r2_c3_e2.png', result)

img = cv2.imread('0150.png')
result = lens_blur(img, radius=2, components=4, exposure_gamma=2)
cv2.imwrite('./0150_lens_r2_c4_e2.png', result)

img = cv2.imread('0150.png')
result = lens_blur(img, radius=2, components=5, exposure_gamma=2)
cv2.imwrite('./0150_lens_r2_c5_e4.png', result)

img = cv2.imread('0150.png')
result = lens_blur(img, radius=2, components=6, exposure_gamma=2)
cv2.imwrite('./0150_lens_r2_c2_e4.png', result)

img = cv2.imread('0150.png')
result = lens_blur(img, radius=2, components=2, exposure_gamma=2)
```

Testing out lens blur strength parameters (radius, components, exposure_gamma)

Testing lens blur radius parameter visualization (components and exposure_gamma constant)

```python
def print_text_to_textfile(file_name, text_to_append):
            file_object.write(text_to_append)


# iterate over files in folder
for filename in os.listdir(input_folder_path):

    # check if image
    if filename.endswith('.png'):

        # construct full input file path
        input_file_path = os.path.join(input_folder_path, filename)

        # read the image using cv2
        img = cv2.imread(input_file_path)

        # selecting a random lens blur radius to adjust the strength of the lens blur degrada
        random_lens_blur_radius = random.randint(2, 3)

        # apply lens blur
        result = lens_blur(img, radius=random_lens_blur_radius, components=4, exposure_gamma=

        # construct full output file path
        output_file_path = os.path.join(output_folder_path, filename)

        # save image in output folder
        cv2.imwrite(output_file_path, result)

        # add degradation strength to degradation output text file
        print_text_to_textfile(os.path.join(textfile_path, textfile_name),filename + ' - ' +

        # print out in console aswell so I see whats happening
        print(filename, ' - lens blur radius:', random_lens_blur_radius)

except:
    print("An error occurred!")
```




```
6383T2.png  - lens blur radius: 3
7806T0.png  - lens blur radius: 2
1549T2.png  - lens blur radius: 3
5311T2.png  - lens blur radius: 2
1136T1.png  - lens blur radius: 2
0718T0.png  - lens blur radius: 3
6212T2.png  - lens blur radius: 3
8359T2.png  - lens blur radius: 2
7369T0.png  - lens blur radius: 3
7151T3.png  - lens blur radius: 3
6214T0.png  - lens blur radius: 3
2632T2.png  - lens blur radius: 2
4446T1.png  - lens blur radius: 3
5144T0.png  - lens blur radius: 3
1035T3.png  - lens blur radius: 3
6662T1.png  - lens blur radius: 2
6792T1.png  - lens blur radius: 3
5898T0.png  - lens blur radius: 3
3655T1.png  - lens blur radius: 2
2313T2.png  - lens blur radius: 2
3380T0.png  - lens blur radius: 3
1773T1.png  - lens blur radius: 2
0987T2.png  - lens blur radius: 3
0626T1.png  - lens blur radius: 3
2278T2.png  - lens blur radius: 2
0337T0.png  - lens blur radius: 3
1035T0.png  - lens blur radius: 2
5143T1.png  - lens blur radius: 3
4082T2.png  - lens blur radius: 2
7528T0.png  - lens blur radius: 3
5773T2.png  - lens blur radius: 2
7024T3.png  - lens blur radius: 2
1356T2.png  - lens blur radius: 3
8207T2.png  - lens blur radius: 2
8054T3.png  - lens blur radius: 3
0915T1.png  - lens blur radius: 2
8260T3.png  - lens blur radius: 3
4841T3.png  - lens blur radius: 2
1653T1.png  - lens blur radius: 3
5194T0.png  - lens blur radius: 3
7747T2.png  - lens blur radius: 2
2678T0.png  - lens blur radius: 3
6009T1.png  - lens blur radius: 3
0979T2.png  - lens blur radius: 3
8014T0.png  - lens blur radius: 2
0377T0.png  - lens blur radius: 3
1424T1.png  - lens blur radius: 3
2025T2.png  - lens blur radius: 2
1659T0.png  - lens blur radius: 3
2557T0.png  - lens blur radius: 2
6660T0.png  - lens blur radius: 3
1079T2.png  - lens blur radius: 2
7684T2.png  - lens blur radius: 2
7435T1.png  - lens blur radius: 3
7789T1.png  - lens blur radius: 2
2079T0.png  - lens blur radius: 2
0260T2.png  - lens blur radius: 2
3257T0.png  - lens blur radius: 3
1047T3.png  - lens blur radius: 3
6753T1.png  - lens blur radius: 3
0177T3.png  - lens blur radius: 2
5117T2.png  - lens blur radius: 2
4179T3.png  - lens blur radius: 3
3535T1.png  - lens blur radius: 3
1053T0.png  - lens blur radius: 2
5056T1.png  - lens blur radius: 3
4172T3.png  - lens blur radius: 3
6014T3.png  - lens blur radius: 2
7086T3.png  - lens blur radius: 3
```

Applying lens blur to the dataset with lens blur radius between 2 and 3 while keeping components at 4 and exposure_gamma at 2.

Processing Step 3: Applying realistic noise

In this step I use my own LUD-VAE model trained on RealLR200 which includes 200 real-world low-resolution images (from the SeeSR github repo) to apply realistic noise to the dataset, with strengths based on my tests.

Since in my previous version of this dataset applied a temperature between 0 and 0.15, all the images had noise to various degrees, but there was not a single image for the network to learn from that had not noise in it, even if it was small (no noise-free lr would exist).

So I am applying the same strategy as with the blur, by duplicating / combining again, and setting a minimum strength to noise that is already noticeable. So there will be noise&blur free images, blurry images, noisy images, and blurry&noisy images in the dataset. Scaling and compression steps are still to follow.

```
 |  0.001 | -0.057 |  0.064 |  0.033 | torch.Size([128]) || proj_c_2.proj.bias
 | -0.000 | -0.062 |  0.062 |  0.036 | torch.Size([128, 256, 1, 1]) || proj_n_2.proj.weight
 | -0.002 | -0.062 |  0.061 |  0.035 | torch.Size([128]) || proj_n_2.proj.bias
 |  0.000 | -0.100 |  0.115 |  0.036 | torch.Size([128, 256, 1, 1]) || proj_1.proj.weight
 |  0.004 | -0.065 |  0.062 |  0.030 | torch.Size([128]) || proj_1.proj.bias
 | -0.000 | -0.137 |  0.112 |  0.037 | torch.Size([128, 256, 1, 1]) || proj_c_1.proj.weight
 | -0.004 | -0.063 |  0.067 |  0.033 | torch.Size([128]) || proj_c_1.proj.bias
 |  0.000 | -0.062 |  0.062 |  0.036 | torch.Size([128, 256, 1, 1]) || proj_n_1.proj.weight
 |  0.005 | -0.062 |  0.061 |  0.036 | torch.Size([128]) || proj_n_1.proj.bias
 |  0.000 | -0.085 |  0.089 |  0.045 | torch.Size([3, 128, 1, 1]) || outconv.weight
 |  0.016 | -0.049 |  0.055 |  0.056 | torch.Size([3]) || outconv.bias

24-02-20 22:41:55.192 : <epoch: 83, iter:  91,000, lr:1.000e-04> loss: 2.136e-04 reconstruction_loss: 2.136e-04 kl_loss: 0.000e+00
24-02-20 22:47:23.235 : <epoch:166, iter:  92,000, lr:1.000e-04> loss: 2.108e-04 reconstruction_loss: 2.108e-04 kl_loss: 0.000e+00
24-02-20 22:52:48.950 : <epoch:249, iter:  93,000, lr:1.000e-04> loss: 2.081e-04 reconstruction_loss: 2.081e-04 kl_loss: 0.000e+00
24-02-20 22:58:14.766 : <epoch:333, iter:  94,000, lr:1.000e-04> loss: 2.049e-04 reconstruction_loss: 2.049e-04 kl_loss: 0.000e+00
24-02-20 23:03:40.196 : <epoch:416, iter:  95,000, lr:1.000e-04> loss: 1.718e-04 reconstruction_loss: 1.718e-04 kl_loss: 0.000e+00
24-02-20 23:09:05.892 : <epoch:499, iter:  96,000, lr:1.000e-04> loss: 1.669e-04 reconstruction_loss: 1.669e-04 kl_loss: 0.000e+00
24-02-20 23:14:31.652 : <epoch:583, iter:  97,000, lr:1.000e-04> loss: 1.593e-04 reconstruction_loss: 1.593e-04 kl_loss: 0.000e+00
24-02-20 23:19:57.196 : <epoch:666, iter:  98,000, lr:1.000e-04> loss: 2.438e-04 reconstruction_loss: 2.438e-04 kl_loss: 0.000e+00
24-02-20 23:25:22.894 : <epoch:749, iter:  99,000, lr:1.000e-04> loss: 2.700e-04 reconstruction_loss: 2.700e-04 kl_loss: 0.000e+00
24-02-20 23:30:48.747 : <epoch:833, iter: 100,000, lr:2.500e-05> loss: 2.155e-04 reconstruction_loss: 2.155e-04 kl_loss: 0.000e+00
24-02-20 23:30:48.747 : Saving the model.
24-02-20 23:36:14.970 : <epoch:916, iter: 101,000, lr:5.000e-05> loss: 2.003e-04 reconstruction_loss: 2.003e-04 kl_loss: 0.000e+00
24-02-20 23:41:40.953 : <epoch:999, iter: 102,000, lr:5.000e-05> loss: 1.787e-04 reconstruction_loss: 1.787e-04 kl_loss: 0.000e+00
24-02-20 23:47:07.072 : <epoch:1083, iter: 103,000, lr:5.000e-05> loss: 1.924e-04 reconstruction_loss: 1.924e-04 kl_loss: 0.000e+00
24-02-20 23:52:33.057 : <epoch:1166, iter: 104,000, lr:5.000e-05> loss: 1.581e-04 reconstruction_loss: 1.581e-04 kl_loss: 0.000e+00
24-02-20 23:57:59.038 : <epoch:1249, iter: 105,000, lr:5.000e-05> loss: 2.214e-04 reconstruction_loss: 2.214e-04 kl_loss: 0.000e+00
24-02-21 00:03:24.934 : <epoch:1333, iter: 106,000, lr:5.000e-05> loss: 1.975e-04 reconstruction_loss: 1.975e-04 kl_loss: 0.000e+00
24-02-21 00:08:50.814 : <epoch:1416, iter: 107,000, lr:5.000e-05> loss: 1.924e-04 reconstruction_loss: 1.924e-04 kl_loss: 0.000e+00
24-02-21 00:14:16.745 : <epoch:1499, iter: 108,000, lr:5.000e-05> loss: 2.096e-04 reconstruction_loss: 2.096e-04 kl_loss: 0.000e+00
24-02-21 00:19:42.895 : <epoch:1583, iter: 109,000, lr:5.000e-05> loss: 1.227e-04 reconstruction_loss: 1.227e-04 kl_loss: 0.000e+00
24-02-21 00:25:08.836 : <epoch:1666, iter: 110,000, lr:5.000e-05> loss: 2.361e-04 reconstruction_loss: 2.361e-04 kl_loss: 0.000e+00
24-02-21 00:25:08.836 : Saving the model.
24-02-21 00:30:34.885 : <epoch:1749, iter: 111,000, lr:5.000e-05> loss: 1.918e-04 reconstruction_loss: 1.918e-04 kl_loss: 0.000e+00
24-02-21 00:36:00.778 : <epoch:1833, iter: 112,000, lr:5.000e-05> loss: 2.590e-04 reconstruction_loss: 2.590e-04 kl_loss: 0.000e+00
24-02-21 00:41:26.775 : <epoch:1916, iter: 113,000, lr:5.000e-05> loss: 1.929e-04 reconstruction_loss: 1.929e-04 kl_loss: 0.000e+00
24-02-21 00:46:52.588 : <epoch:1999, iter: 114,000, lr:5.000e-05> loss: 1.617e-04 reconstruction_loss: 1.617e-04 kl_loss: 0.000e+00
24-02-21 00:52:18.664 : <epoch:2083, iter: 115,000, lr:5.000e-05> loss: 2.019e-04 reconstruction_loss: 2.019e-04 kl_loss: 0.000e+00
24-02-21 00:57:44.596 : <epoch:2166, iter: 116,000, lr:5.000e-05> loss: 2.224e-04 reconstruction_loss: 2.224e-04 kl_loss: 0.000e+00
24-02-21 01:03:10.293 : <epoch:2249, iter: 117,000, lr:5.000e-05> loss: 2.079e-04 reconstruction_loss: 2.079e-04 kl_loss: 0.000e+00
24-02-21 01:08:36.259 : <epoch:2333, iter: 118,000, lr:5.000e-05> loss: 1.794e-04 reconstruction_loss: 1.794e-04 kl_loss: 0.000e+00
24-02-21 01:14:02.000 : <epoch:2416, iter: 119,000, lr:5.000e-05> loss: 1.510e-04 reconstruction_loss: 1.510e-04 kl_loss: 0.000e+00
24-02-21 01:19:27.851 : <epoch:2499, iter: 120,000, lr:5.000e-05> loss: 2.137e-04 reconstruction_loss: 2.137e-04 kl_loss: 0.000e+00
24-02-21 01:19:27.851 : Saving the model.
24-02-21 01:24:54.321 : <epoch:2583, iter: 121,000, lr:5.000e-05> loss: 1.754e-04 reconstruction_loss: 1.754e-04 kl_loss: 0.000e+00
24-02-21 01:30:20.100 : <epoch:2666, iter: 122,000, lr:5.000e-05> loss: 1.943e-04 reconstruction_loss: 1.943e-04 kl_loss: 0.000e+00
24-02-21 01:35:46.045 : <epoch:2749, iter: 123,000, lr:5.000e-05> loss: 1.956e-04 reconstruction_loss: 1.956e-04 kl_loss: 0.000e+00
24-02-21 01:41:12.123 : <epoch:2833, iter: 124,000, lr:5.000e-05> loss: 1.664e-04 reconstruction_loss: 1.664e-04 kl_loss: 0.000e+00
24-02-21 01:46:37.944 : <epoch:2916, iter: 125,000, lr:5.000e-05> loss: 2.182e-04 reconstruction_loss: 2.182e-04 kl_loss: 0.000e+00
24-02-21 01:52:03.995 : <epoch:2999, iter: 126,000, lr:5.000e-05> loss: 2.016e-04 reconstruction_loss: 2.016e-04 kl_loss: 0.000e+00
24-02-21 01:57:29.984 : <epoch:3083, iter: 127,000, lr:5.000e-05> loss: 1.962e-04 reconstruction_loss: 1.962e-04 kl_loss: 0.000e+00
24-02-21 02:02:55.935 : <epoch:3166, iter: 128,000, lr:5.000e-05> loss: 1.891e-04 reconstruction_loss: 1.891e-04 kl_loss: 0.000e+00
24-02-21 02:08:21.774 : <epoch:3249, iter: 129,000, lr:5.000e-05> loss: 1.874e-04 reconstruction_loss: 1.874e-04 kl_loss: 0.000e+00
24-02-21 02:13:47.869 : <epoch:3333, iter: 130,000, lr:5.000e-05> loss: 1.851e-04 reconstruction_loss: 1.851e-04 kl_loss: 0.000e+00
24-02-21 02:13:47.869 : Saving the model.
24-02-21 02:19:13.724 : <epoch:3416, iter: 131,000, lr:5.000e-05> loss: 1.649e-04 reconstruction_loss: 1.649e-04 kl_loss: 0.000e+00
24-02-21 02:24:39.884 : <epoch:3499, iter: 132,000, lr:5.000e-05> loss: 2.219e-04 reconstruction_loss: 2.219e-04 kl_loss: 0.000e+00
24-02-21 02:30:06.182 : <epoch:3583, iter: 133,000, lr:5.000e-05> loss: 2.081e-04 reconstruction_loss: 2.081e-04 kl_loss: 0.000e+00
24-02-21 02:35:32.167 : <epoch:3666, iter: 134,000, lr:5.000e-05> loss: 2.379e-04 reconstruction_loss: 2.379e-04 kl_loss: 0.000e+00
24-02-21 02:40:58.110 : <epoch:3749, iter: 135,000, lr:5.000e-05> loss: 2.044e-04 reconstruction_loss: 2.044e-04 kl_loss: 0.000e+00
24-02-21 02:46:24.362 : <epoch:3833, iter: 136,000, lr:5.000e-05> loss: 2.290e-04 reconstruction_loss: 2.290e-04 kl_loss: 0.000e+00
24-02-21 02:51:50.340 : <epoch:3916, iter: 137,000, lr:5.000e-05> loss: 2.270e-04 reconstruction_loss: 2.270e-04 kl_loss: 0.000e+00
24-02-21 02:57:16.330 : <epoch:3999, iter: 138,000, lr:5.000e-05> loss: 1.800e-04 reconstruction_loss: 1.800e-04 kl_loss: 0.000e+00
24-02-21 03:02:42.422 : <epoch:4083, iter: 139,000, lr:5.000e-05> loss: 2.069e-04 reconstruction_loss: 2.069e-04 kl_loss: 0.000e+00
```

2 nights, ~ 17 hours,
190’000 iters

LUD-VAE training of my ludvae200 degradation model
on RealLR200 (which includes 200 real-world low-resolution images) provided on the SeeSR github repo.

Ludvae200 degraded image to showcase realistically added noise of my model

Ludvae200 degraded image to showcase realistically added noise of my model

```python
        data = file_object.read(100)
        if len(data) > 0:
            file_object.write("\n")
        # Append text at the end of file
        file_object.write(text_to_append)

# ----------------------------------------
# load model
# ----------------------------------------

model = LUDVAE()
states = torch.load(model_path)

model.load_state_dict(states, strict=True)
model.eval()

for k, v in model.named_parameters():
    v.requires_grad = False

model = model.to(device)

H_paths = util.get_image_paths(H_path)

# I set these strength settings for noise and temperature based on tests with the ludvae200 model I trained specifically
# since I will have non-processed noise-free images in the dataset (since this would never reach noise purely 0, so a non-noise image would be non-existent in the dataset,
# maybe reach very small noise but never none), i increased these values a bit to cut out too low noise degraded images
# min: noise 5 with temperature 0.03
# max: noise 10 with temperature 0.2 (should be enough as an upper limit, increased from previous try)

for idx, img in enumerate(H_paths):

    # ------------------------------------
    # (1) img_H
    # ------------------------------------

    img_name, ext = os.path.splitext(os.path.basename(img))
    img_H = util.imread_uint(img, n_channels=3)
    img_hH = img_H.copy()

    img_H = util.uint2tensor4(img_H).to(device)
    img_hH = util.uint2tensor4(img_hH).to(device)
    noise_strength = uniform(5,10)
    img_hH = img_hH + torch.randn_like(img_hH) * noise_strength / 255.0

    # ------------------------------------
    # (2) img_G
    # ------------------------------------

    label_H = torch.zeros(1, 1, 1, 1).long().to(device)
    temperature_strength = uniform(0.03,0.2)
    img_G = model.translate(img_H, img_hH, label_H, temperature=temperature_strength)
    img_G = util.tensor2uint(img_G)

    # ------------------------------------
    # save results
    # ------------------------------------

    util.imsave(img_G, os.path.join(G_path, img_name + ext))

    # ------------------------------------
    # log degradation strength
    # ------------------------------------

    # add degradation strength to degradation output text file
    print_text_to_textfile(os.path.join(textfile_path, textfile_name),img_name + ' - noise: ' + str(noise_strength) + ', temperature: ' + str(temperature_strength))

    # print out in console aswell so I see whats happening
    print(img_name + ' - noise: ' + str(noise_strength) + ', temperature: ' + str(temperature_strength))

if __name__ == '__main__':

    main()
```

```
1016 - noise: 6.53380213924O632, temperature: 0.135579086956494
10160 - noise: 8.455503708108385, temperature: 0.053746467305090084
10161 - noise: 7.01318864618761, temperature: 0.12468049461062762
10162 - noise: 8.378534806831336, temperature: 0.11020570144887328
10163 - noise: 6.451089535259765, temperature: 0.034978947263060625
10164 - noise: 6.164043063395509, temperature: 0.1322323642758328
10165 - noise: 9.34577883438355, temperature: 0.13342481448650073
10166 - noise: 5.630130874844354, temperature: 0.04265475019412145
10167 - noise: 8.350477474631694, temperature: 0.15929992576380425
10168 - noise: 5.886101875618746, temperature: 0.18343099578325583
10169 - noise: 8.453340263100529, temperature: 0.11865464611267179
1017 - noise: 5.698272950744755, temperature: 0.07450988645807916
10170 - noise: 7.335922264316634, temperature: 0.19029835358758554
10171 - noise: 6.545947324805594, temperature: 0.13212944412741223
10172 - noise: 7.740129601979456, temperature: 0.09258503007685474
10173 - noise: 5.887457725321606, temperature: 0.09895086141117512
10174 - noise: 7.834664485363965, temperature: 0.03415154692715354
10175 - noise: 6.159411407657588, temperature: 0.13803272652447623
10176 - noise: 7.081033217327755, temperature: 0.1969034287935458
10177 - noise: 9.815845491445351, temperature: 0.03801077239107685
10178 - noise: 6.706275413440608, temperature: 0.04916995468897137
10179 - noise: 5.135448736290073, temperature: 0.08128689148875234
1018 - noise: 5.961447297226833, temperature: 0.075813376571823
10180 - noise: 5.763750186148448, temperature: 0.1780158286415424
10181 - noise: 8.600831287662553, temperature: 0.04069908573135779
10182 - noise: 9.998268504696036, temperature: 0.05441296429883806
10183 - noise: 7.719467803389598, temperature: 0.1651832473675164
10184 - noise: 6.0299792069692515, temperature: 0.07317197581663218
10185 - noise: 8.426857123166752, temperature: 0.18484914227070387
10186 - noise: 9.863585266317598, temperature: 0.1218735934647855
10187 - noise: 9.278934684455354, temperature: 0.11095652022099373
10188 - noise: 8.786813265625039, temperature: 0.13118657215404098
10189 - noise: 8.502390815234522, temperature: 0.18091869215683792
1019 - noise: 6.6920704829248585, temperature: 0.18397081425208092
10190 - noise: 5.595025998106429, temperature: 0.15145248411988538
10191 - noise: 5.100390219651978, temperature: 0.05205779139002355
10192 - noise: 6.880509168294787, temperature: 0.15163122480631844
10193 - noise: 6.633545339138198, temperature: 0.04654439553180263
10194 - noise: 9.422885227209612, temperature: 0.06729674359189786
10195 - noise: 6.468189180212968, temperature: 0.08360592002751446
10196 - noise: 7.964259467912977, temperature: 0.14636610671134354
10197 - noise: 5.96663273819735, temperature: 0.15295337268102643
10198 - noise: 7.019854663081286, temperature: 0.11643620032897752
10199 - noise: 6.6219211755651575, temperature: 0.06653098970816079
102 - noise: 6.727037720328608, temperature: 0.07747384637378193
1020 - noise: 5.2575466183018476, temperature: 0.08657119618852449
10200 - noise: 5.794465054480316, temperature: 0.13247516069025303
10201 - noise: 6.990758772796733, temperature: 0.09757458082042435
10202 - noise: 5.968983266610068, temperature: 0.05395819294416622
10203 - noise: 5.897872996374947, temperature: 0.08083120933482217
10204 - noise: 7.73043988559928, temperature: 0.08167187515453109
10205 - noise: 5.514308533445452, temperature: 0.15016137725123468
10206 - noise: 7.046618768050461, temperature: 0.0390187592305896
10207 - noise: 5.2992275081098175, temperature: 0.030176261649571056
10208 - noise: 6.711508032479756, temperature: 0.10341558469331316
10209 - noise: 7.084774158269541, temperature: 0.087982028873095
1021 - noise: 8.007014867305877, temperature: 0.13430430722508402
10210 - noise: 5.9363951195157005, temperature: 0.1222778974983491
10211 - noise: 6.399607028961906, temperature: 0.12645071995965196
10212 - noise: 8.769300360238088, temperature: 0.03347614559337842
10213 - noise: 5.286629132737009, temperature: 0.0530002097328173
10214 - noise: 9.885783190724572, temperature: 0.11221598319760677
10215 - noise: 8.423072549395052, temperature: 0.17111076602680533
10216 - noise: 6.871482487486145, temperature: 0.19088536821324734
10217 - noise: 5.126099814599729, temperature: 0.14279763615486812
10218 - noise: 9.215843358952966, temperature: 0.18395358983781893
10219 - noise: 8.039824131977335, temperature: 0.1407523610663915
1022 - noise: 5.095298129870817, temperature: 0.17907187230064567
10220 - noise: 7.917386205331605, temperature: 0.033932204401454676
10221 - noise: 6.771320724590104, temperature: 0.07043761420744576
10222 - noise: 6.2203193802088315, temperature: 0.18560007029320566
10223 - noise: 8.252920142629282, temperature: 0.1775386362540875
```

Adding realistic noise with my ludvae200 degradation model to the dataset

Just to showcase, I currently have 4 degraded versions per image on a per-scale basis (example scale 1):

Blur & Noise (top left)

Blur (top right)

Noise (bottom left)

None (bottom right)

(so since we had multiscaled with 4 scales in this way we turned each image of the original dataset into 16 images)

Processing Step 4: Applying scale and compression

In this step, we use kim's Dataset Destroyer to first scale then compress in a 2-step manner, both times scaling with 0.5 to reach a x4 lr output for the 4x paired dataset.

Scaling algos used: down_up, nearest, linear, cubic_catrom, cubic_mitchell, cubic_bspline, lanczos, gaussian

Compression: jpg 40-100 and webp 45-100

Strengths are applied in a randomized manner, since we have 135'872 lr images now (multiscaling and degradation versions) this should help having a good distribution of applied compression strengths.

```
# Compression settings
[compression]
# List of available compression algorithms (e.g., mpeg,mpeg2,h264,hevc,jpeg,webp,vp9)
# Using more intensive codecs (such as vp9) in combination with other degradations may result in ffmpeg errors
algorithms = jpeg,webp
# Whether to choose a random algorithm from the list
randomize = True
# JPEG Quality Levels
jpeg_quality_range = 40, 100
# WebP Quality levels
webp_quality_range = 45, 100
# H.264 video quality levels in CRF format
h264_crf_level_range = 23,32
# HEVC video quality levels in CRF format
hevc_crf_level_range = 25,34
# VP9 video quality levels in CRF format
vp9_crf_level_range = 25,35
# Quality control for MPEG codec. Range 1-31
mpeg_qscale_range = 2,15
# Quality  control for MPEG2 codec. Range 1-31
mpeg2_qscale_range = 2,15

# Scale settings
[scale]
# List of available scale algorithms (e.g., down_up,nearest,linear,cubic_catrom,cubic_mitchell,cubic_bspline,lanczos,
algorithms = down_up,nearest,linear,cubic_catrom,cubic_mitchell,cubic_bspline,lanczos,gauss
# List of available scale algorithms when applying down_up
down_up_algorithms = nearest,linear,cubic_catrom,cubic_mitchell,cubic_bspline,lanczos,gauss
# Whether to choose a random scale algorithm each time (True or False)
randomize = True
# Factor to scale your images to (e.g., 0.25, 0.50, 0.75) (0.25 = 25%, 0.50 = 50%)
size_factor = 0.5
# Range of values for down_up (e.g., 0.5,2.0) (0.5 = 50%, 2.0 = 200%)
range = 0.15,1.5
```

```
16683 (1).png - scale: nearest size factor=0.5, compression: jpeg quality=46
36086 (1).png - scale: gauss size factor=0.5, compression: webp quality=79
26542 (1).png - scale: cubic_mitchell size factor=0.5, compression: webp quality=76
47163 (1).png - scale: cubic_catrom size factor=0.5, compression: webp quality=47
53592.png - scale: nearest size factor=0.5, compression: webp quality=61
11010.png - scale: cubic_mitchell size factor=0.5, compression: webp quality=52
61204 (1).png - scale: down_up scale1factor=0.69 scale1algorithm=linear
scale2factor=0.72 scale2algorithm=lanczos, compression: jpeg quality=75
58690.png - scale: lanczos size factor=0.5, compression: jpeg quality=66
33831 (1).png - scale: cubic_bspline size factor=0.5, compression: jpeg quality=42
45949.png - scale: cubic_mitchell size factor=0.5, compression: webp quality=52
21970 (1).png - scale: cubic_bspline size factor=0.5, compression: webp quality=86
274.png - scale: gauss size factor=0.5, compression: jpeg quality=68
31676.png - scale: nearest size factor=0.5, compression: jpeg quality=97
55219.png - scale: gauss size factor=0.5, compression: webp quality=100
53927 (1).png - scale: gauss size factor=0.5, compression: webp quality=79
32080.png - scale: linear size factor=0.5, compression: webp quality=67
54156 (1).png - scale: gauss size factor=0.5, compression: webp quality=74
62525.png - scale: linear size factor=0.5, compression: webp quality=63
2750.png - scale: cubic_bspline size factor=0.5, compression: jpeg quality=62
32105.png - scale: cubic_mitchell size factor=0.5, compression: webp quality=97
26340 (1).png - scale: cubic_bspline size factor=0.5, compression: jpeg quality=70
65737 (1).png - scale: cubic_bspline size factor=0.5, compression: jpeg quality=75
26866.png - scale: cubic_catrom size factor=0.5, compression: jpeg quality=80
67663 (1).png - scale: gauss size factor=0.5, compression: jpeg quality=79
22320 (1).png - scale: cubic_catrom size factor=0.5, compression: jpeg quality=64
62492 (1).png - scale: cubic_catrom size factor=0.5, compression: jpeg quality=82
44725 (1).png - scale: cubic_mitchell size factor=0.5, compression: jpeg quality=69
39567 (1).png - scale: cubic_bspline size factor=0.5, compression: webp quality=75
50231 (1).png - scale: gauss size factor=0.5, compression: webp quality=78
20157.png - scale: linear size factor=0.5, compression: jpeg quality=40
23695 (1).png - scale: gauss size factor=0.5, compression: webp quality=54
4512.png - scale: cubic_catrom size factor=0.5, compression: webp quality=78
58505 (1).png - scale: cubic_mitchell size factor=0.5, compression: webp quality=67
34152 (1).png - scale: cubic_bspline size factor=0.5, compression: jpeg quality=69
51838.png - scale: lanczos size factor=0.5, compression: webp quality=100
30477.png - scale: cubic_catrom size factor=0.5, compression: webp quality=70
49976.png - scale: cubic_catrom size factor=0.5, compression: webp quality=99
57568.png - scale: linear size factor=0.5, compression: webp quality=67
53727.png - scale: cubic_catrom size factor=0.5, compression: jpeg quality=66
34973 (1).png - scale: nearest size factor=0.5, compression: webp quality=83
47597.png - scale: gauss size factor=0.5, compression: jpeg quality=75
45871.png - scale: lanczos size factor=0.5, compression: webp quality=71
21739.png - scale: lanczos size factor=0.5, compression: webp quality=94
44876.png - scale: nearest size factor=0.5, compression: jpeg quality=84
45782 (1).png - scale: cubic_bspline size factor=0.5, compression: webp quality=98
12601 (1).png - scale: nearest size factor=0.5, compression: webp quality=68
47172.png - scale: cubic_catrom size factor=0.5, compression: jpeg quality=53
33818.png - scale: cubic_catrom size factor=0.5, compression: webp quality=81
52019 (1).png - scale: gauss size factor=0.5, compression: jpeg quality=50
27037.png - scale: linear size factor=0.5, compression: jpeg quality=74
33694.png - scale: cubic_bspline size factor=0.5, compression: webp quality=75
28292 (1).png - scale: cubic_bspline size factor=0.5, compression: jpeg quality=53
23419 (1).png - scale: linear size factor=0.5, compression: webp quality=86
38452.png - scale: cubic_mitchell size factor=0.5, compression: jpeg quality=78
60036.png - scale: gauss size factor=0.5, compression: webp quality=88
5556 (1).png - scale: nearest size factor=0.5, compression: webp quality=80
57098.png - scale: cubic_catrom size factor=0.5, compression: jpeg quality=63
66170 (1).png - scale: gauss size factor=0.5, compression: webp quality=93
33961 (1).png - scale: cubic_mitchell size factor=0.5, compression: jpeg quality=61
10699.png - scale: cubic_bspline size factor=0.5, compression: webp quality=96
11849.png - scale: cubic_bspline size factor=0.5, compression: jpeg quality=88
39520 (1).png - scale: linear size factor=0.5, compression: jpeg quality=74
23871.png - scale: linear size factor=0.5, compression: webp quality=73
36539.png - scale: lanczos size factor=0.5, compression: jpeg quality=72
11424 (1).png - scale: cubic_bspline size factor=0.5, compression: jpeg quality=41
39916.png - scale: lanczos size factor=0.5, compression: webp quality=66
18462.png - scale: nearest size factor=0.5, compression: webp quality=75
```

Applying scale and compression

```
# Compression settings
[compression]
# List of available compression algorithms (e.g., mpeg,mpeg2,h264,hevc,jpeg,webp,vp9)
# Using more intensive codecs (such as vp9) in combination with other degradations may result in ffmpeg errors
algorithms = jpeg,webp
# Whether to choose a random algorithm from the list
randomize = True
# JPEG Quality Levels
jpeg_quality_range = 40, 100
# WebP Quality levels
webp_quality_range = 45, 100
# H.264 video quality levels in CRF format
h264_crf_level_range = 23,32
# HEVC video quality levels in CRF format
hevc_crf_level_range = 25,34
# VP9 video quality levels in CRF format
vp9_crf_level_range = 25,35
# Quality control for MPEG codec. Range 1-31
mpeg_qscale_range = 2,15
# Quality  control for MPEG2 codec. Range 1-31
mpeg2_qscale_range = 2,15

# Scale settings
[scale]
# List of available scale algorithms (e.g., down_up,nearest,linear,cubic_catrom,cubic_mitchell,cubic_bspline,lanczos,g
algorithms = down_up,nearest,linear,cubic_catrom,cubic_mitchell,cubic_bspline,lanczos,gauss
# List of available scale algorithms when applying down_up
down_up_algorithms = nearest,linear,cubic_catrom,cubic_mitchell,cubic_bspline,lanczos,gauss
# Whether to choose a random scale algorithm each time (True or False)
randomize = True
# Factor to scale your images to (e.g., 0.25, 0.50, 0.75) (0.25 = 25%, 0.50 = 50%)
size_factor = 0.5
# Range of values for down_up (e.g., 0.5,2.0) (0.5 = 50%, 2.0 = 200%)
range = 0.15,1.5
```

```
47163 (1).png - scale: cubic_bspline size factor=0.5, compression: jpeg quality=47
53592.png - scale: lanczos size factor=0.5, compression: webp quality=53
26542 (1).png - scale: linear size factor=0.5, compression: webp quality=86
61204 (1).png - scale: cubic_bspline size factor=0.5, compression: jpeg quality=84
36086 (1).png - scale: down_up scale1factor=0.96 scale1algorithm=cubic_bspline
scale2factor=0.52 scale2algorithm=nearest, compression: jpeg quality=41
16683 (1).png - scale: nearest size factor=0.5, compression: webp quality=52
58690.png - scale: down_up scale1factor=0.21 scale1algorithm=lanczos
scale2factor=2.35 scale2algorithm=lanczos, compression: jpeg quality=50
11010.png - scale: down_up scale1factor=0.26 scale1algorithm=linear scale2factor=1.91
scale2algorithm=gauss, compression: webp quality=86
33831 (1).png - scale: down_up scale1factor=1.03 scale1algorithm=cubic_catrom
scale2factor=0.48 scale2algorithm=nearest, compression: webp quality=51
45949.png - scale: cubic_catrom size factor=0.5, compression: webp quality=87
55219.png - scale: gauss size factor=0.5, compression: jpeg quality=88
53927 (1).png - scale: down_up scale1factor=0.96 scale1algorithm=gauss
scale2factor=0.52 scale2algorithm=cubic_mitchell, compression: jpeg quality=69
31676.png - scale: down_up scale1factor=1.20 scale1algorithm=linear scale2factor=0.42
scale2algorithm=lanczos, compression: jpeg quality=71
274.png - scale: cubic_catrom size factor=0.5, compression: jpeg quality=69
32080.png - scale: cubic_catrom size factor=0.5, compression: jpeg quality=100
21970 (1).png - scale: gauss size factor=0.5, compression: jpeg quality=51
32105.png - scale: cubic_bspline size factor=0.5, compression: jpeg quality=45
26340 (1).png - scale: linear size factor=0.5, compression: webp quality=53
62525.png - scale: nearest size factor=0.5, compression: jpeg quality=98
54156 (1).png - scale: linear size factor=0.5, compression: webp quality=96
22320 (1).png - scale: nearest size factor=0.5, compression: jpeg quality=76
2750.png - scale: lanczos size factor=0.5, compression: webp quality=100
26866.png - scale: cubic_bspline size factor=0.5, compression: webp quality=100
65737 (1).png - scale: lanczos size factor=0.5, compression: webp quality=80
67663 (1).png - scale: down_up scale1factor=0.38 scale1algorithm=cubic_mitchell
scale2factor=1.33 scale2algorithm=nearest, compression: webp quality=79
50231 (1).png - scale: gauss size factor=0.5, compression: webp quality=55
44725 (1).png - scale: linear size factor=0.5, compression: jpeg quality=74
23695 (1).png - scale: cubic_catrom size factor=0.5, compression: jpeg quality=91
58505 (1).png - scale: cubic_catrom size factor=0.5, compression: jpeg quality=72
62492 (1).png - scale: cubic_bspline size factor=0.5, compression: webp quality=80
39567 (1).png - scale: linear size factor=0.5, compression: jpeg quality=75
20157.png - scale: lanczos size factor=0.5, compression: jpeg quality=45
4512.png - scale: linear size factor=0.5, compression: jpeg quality=100
51838.png - scale: gauss size factor=0.5, compression: jpeg quality=68
30477.png - scale: cubic_catrom size factor=0.5, compression: jpeg quality=84
49976.png - scale: linear size factor=0.5, compression: webp quality=90
53727.png - scale: gauss size factor=0.5, compression: webp quality=96
34152 (1).png - scale: linear size factor=0.5, compression: jpeg quality=77
57568.png - scale: lanczos size factor=0.5, compression: webp quality=48
47597.png - scale: lanczos size factor=0.5, compression: webp quality=87
34973 (1).png - scale: linear size factor=0.5, compression: jpeg quality=42
45782 (1).png - scale: cubic_catrom size factor=0.5, compression: jpeg quality=89
47172.png - scale: cubic_bspline size factor=0.5, compression: webp quality=65
44876.png - scale: gauss size factor=0.5, compression: jpeg quality=83
12601 (1).png - scale: cubic_mitchell size factor=0.5, compression: webp quality=49
21739.png - scale: cubic_mitchell size factor=0.5, compression: webp quality=99
33818.png - scale: cubic_catrom size factor=0.5, compression: jpeg quality=73
45871.png - scale: cubic_mitchell size factor=0.5, compression: jpeg quality=96
28292 (1).png - scale: down_up scale1factor=1.10 scale1algorithm=nearest
scale2factor=0.45 scale2algorithm=cubic_catrom, compression: jpeg quality=84
23419 (1).png - scale: cubic_bspline size factor=0.5, compression: jpeg quality=81
33694.png - scale: cubic_catrom size factor=0.5, compression: webp quality=83
38452.png - scale: lanczos size factor=0.5, compression: jpeg quality=92
27037.png - scale: down_up scale1factor=0.50 scale1algorithm=linear scale2factor=1.00
scale2algorithm=cubic_catrom, compression: jpeg quality=51
57098.png - scale: cubic_catrom size factor=0.5, compression: jpeg quality=50
52019 (1).png - scale: cubic_bspline size factor=0.5, compression: webp quality=72
66170 (1).png - scale: gauss size factor=0.5, compression: jpeg quality=69
39520 (1).png - scale: nearest size factor=0.5, compression: webp quality=59
10699.png - scale: down_up scale1factor=1.11 scale1algorithm=linear scale2factor=0.45
scale2algorithm=nearest, compression: jpeg quality=43
60036.png - scale: gauss size factor=0.5, compression: webp quality=55
33961 (1).png - scale: lanczos size factor=0.5, compression: webp quality=65
5556 (1).png - scale: down_up scale1factor=0.19 scale1algorithm=gauss
```

Applying re-scaling and re-compression

Final lr’s after (re)scaling and (re)compressing

- same example as previously

(enlarged to fill slide for better visibility)

Scale and compression variants

When looking at the results, all of the lr's will be rescaled and recompressed (look pretty degraded) to various degrees of strengths, but there currently is no non-compressed image in the dataset.

So like I previously made variants for both blur and noise to also have for example non-degraded images in the dataset so the network can learn from a better distribution meaning non-degraded as well as degraded images, I also decided to apply the same strategy here.

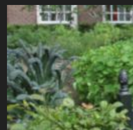
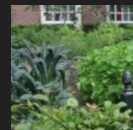
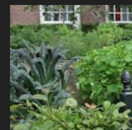
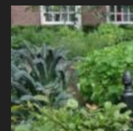
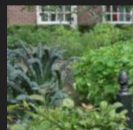
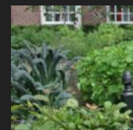
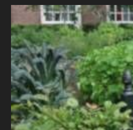
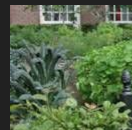
Also made versions and combined again

Only x4 scaled (learn different scaling algorithms on blur&noise, blur, noise and non-degraded images)

X4 scaled and compressed (learn additionally jpg and webp compression to the above)

Rescaled (0.5 scale, then compression, then 0.5 rescale) (learn scaled compression to the above)

Rescaled and recompressed (0.5 scale, compression, 0.5 rescale, recompression) (learn recompression to the above)

	Good Quality	Blurry&Noisy	Blurry	Noisy
x4 downsized only (down_up, nearest, linear, cubic_catrom, cubic_mitchell, cubic_bspline, lanczos, gaussian)				
x4 downsized and compressed (jpg 40-100 or webp 45-100) - uploaded to the web				
x2 downsized, compressed, x2 re-downsized (so contains scaled compression)				
x2 downsized, compressed, x2 re-downsized, recompressed - downloaded and re-uploaded from and to the web				

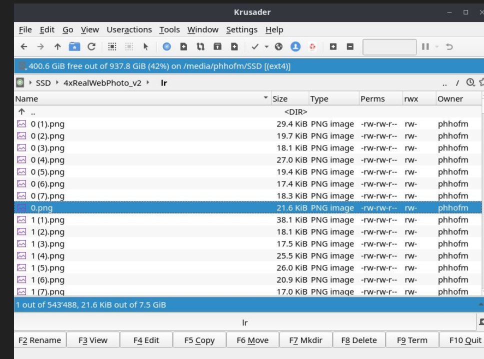
All use cases a sisr model trained on this dataset could be able to handle since I created all these different variants in the lr

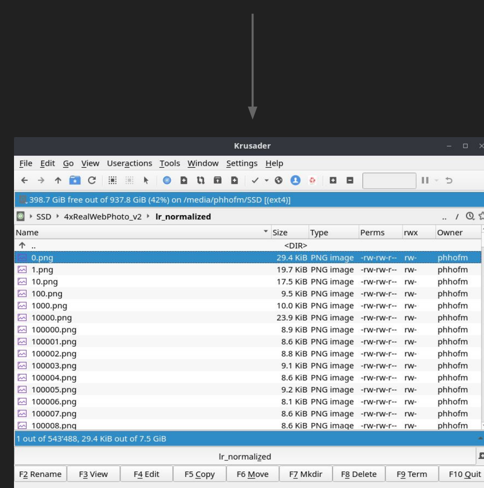
chaiNNer - Untitled •

chaiNNer Alpha v0.21.2

LOAD IMAGES
409352/543488 (75%) ETA: in 33 minutes
Directory: /media/phhofm/SSD/4xReal...
Use WCMatch glob expression
Recursive
Use limit
Stop on first error
512x512 RGB Image
...WebPhoto_v2/hr Directory
Subdirectory Path
51121 (5) Name
408972 Index

SAVE IMAGE
Image
Directory: /media/phhofm/SSD/4xR...
Subdirectory Path optional
Image Name: 408972
Image Format: PNG
Color Depth: 8 Bits/Channel





Normalizing filenames

Dataset stats

Original input dataset: nomos8k - 8’492 images, 6.7 GB

Output dataset: 4xRealWebPhoto_v2 - 1’086’978 images (543’489 image pairs), 132.7 GB

(Disk size could be decreased by removing duplicates from hr and explicitly mapping each lr to its corresponding hr in a many-to-one relationship)