Generating Natural images with direct patch Distributions Matching
Supplementary material

Ariel Elnekave and Yair Weiss
Hebrew University Jerusalem
{Ariel.Elnekave, Yair.Weiss}@mail.huji.ac.il

This document contains supplementary material on our method for Generating natural images by Patch Distribution Matching (GPDM).

1 Python implementation

Our python implementation of GPDM is available at
https://github.com/ariel415el/GPDM.
All GPDM results generated in this document and in the main paper are generated via the python scripts in the 'scripts' sub-folder. Please Consult the README.md file for additional instructions.

All our experiments with GPNN were done with our implementation available at https://github.com/ariel415el/Efficient-GPNN.

2 Method parameters description

In this section we describe our method's configurations in more details As detailed in the paper. GPDM is an iterative multi-scale process for generating images and has multiple configurations controlling the output given the inputs.

pyramid_factor and coarse_dimension: These parameters control the number and sizes of the different scales in which the GPDM works. The input image is repeatedly scaled down by **pyramid_factor** until scaling it will result in one of its dimensions to be less than **coarse_dimension**. All the intermediate images are used as the multi scale pyramid on which the algorithm works.

scale_factors: This parameter defines the aspect ratio of the output. It is relevant mostly for the texture synthesis and retargeting tasks as in all other tasks the output image should have the same size as the input.

init_mode and noise sigma: These parameters effect the first initial guess of the algorithm. **init_mode** can instruct the algorithm to start the optimization from a blank image, from the target image or from another image. **noise_sigma** defines the amount of Gaussian pixel noise added to the initial guess in order to increase variability.

learning_rate and num_optimization_steps: These parameters effect the optimization of the synthesis image at each scale. specifically it defines the number of SGD steps and the step size used for optimizing SWD between the images.

patch_size, stride and num_projections: These parameters control the SWD computation between patches in two images. **patch_size** and **stride** control the way patches are extracted from an image. **num_projection** defines the number of projections used to estimate SWD.

3 Task specific configurations and results

In this section we describe the actual configurations we used for each of the tasks we talk about in the paper. We also add additional result images for each task generated with the noted configuration.

3.1 Image reshuffling

For the task of reshuffling images from the SIGD16, Places50 datasets we used the following configuration:
pyramid_factor=0.85, coarse_dim=28, scale_factors=(1,1), init_mode='zeros', noise_sigma=1.5, patch_size=7, stride=1, num_projections=64, learning_rate=0.01, num_optimization_steps=300

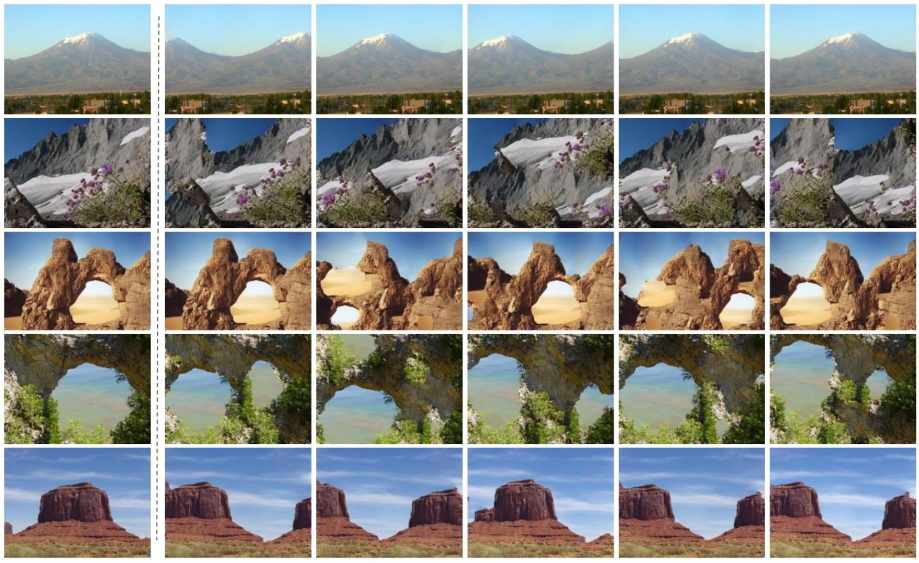

Fig. 1. Reshuffling of images from the SIGD15 dataset. Inputs on the left.

3.2 Image retargeting

As mentioned before, unlike in reshuffling, retargeting generates an output in different size than the output. For that purpose we set the scale-factor parameter as desired, i.e (1,2) for an output which is twice as wide from the input.

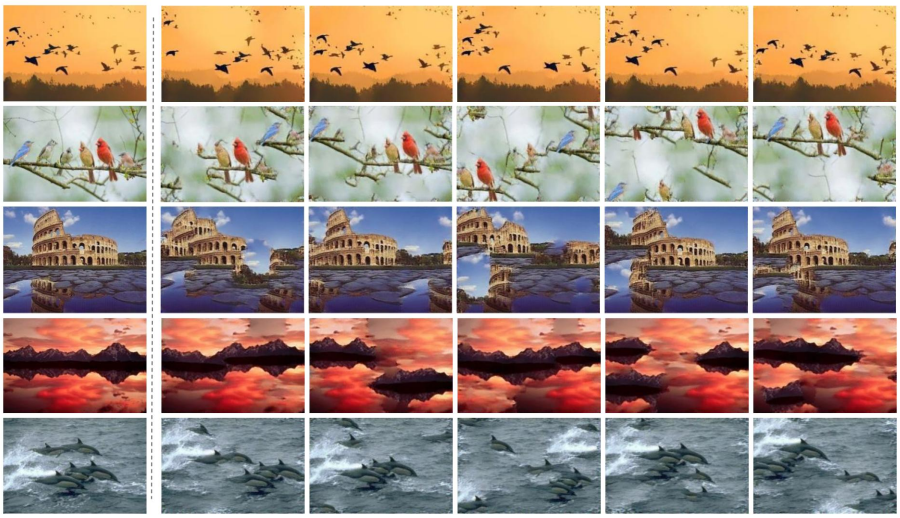

Fig. 2. Reshuffling of images from the Places50 dataset. Inputs on the left.

Another difference in the retargeting task is that we aim for less variability and more coherence in the output. For that reason we start the optimization from an image resized to a different aspect ratio defined by the parameter scale-factor, blur it, (init_mode='blurred_target') and add no pixel noise (noise_sigma=0). Figures [3, 5, 4] show some additional retargeting results. Each batch of images shows the input on the upper left and results where the scale factor parameter is set to (2,2) (1,2), (2,1) in a clockwise order from it. The other parameters used for creating these images are:
pyramid_factor=0.85, coarse_dim=35
init_mode='blurred_target', noise_sigma=0, patch_size=7, stride=1,
num_projections=64, learning_rate=0.01,
num_optimization_steps=300

Fig. 3. Retargeting of famous paintings.

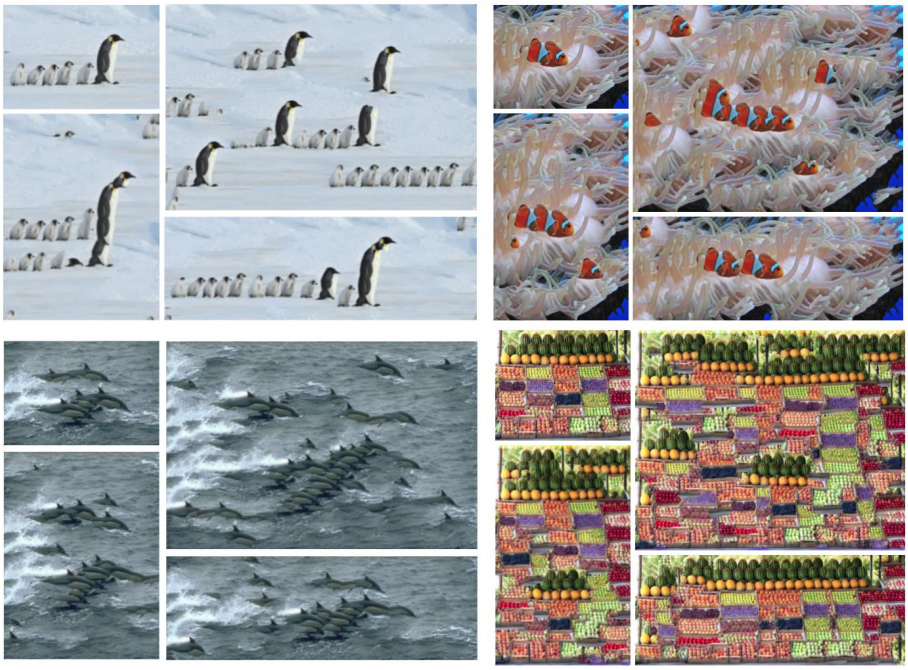

Fig. 4. Retargeting of some selected images from previous papers.

Fig. 5. Less sucsessfull examples of retargeting

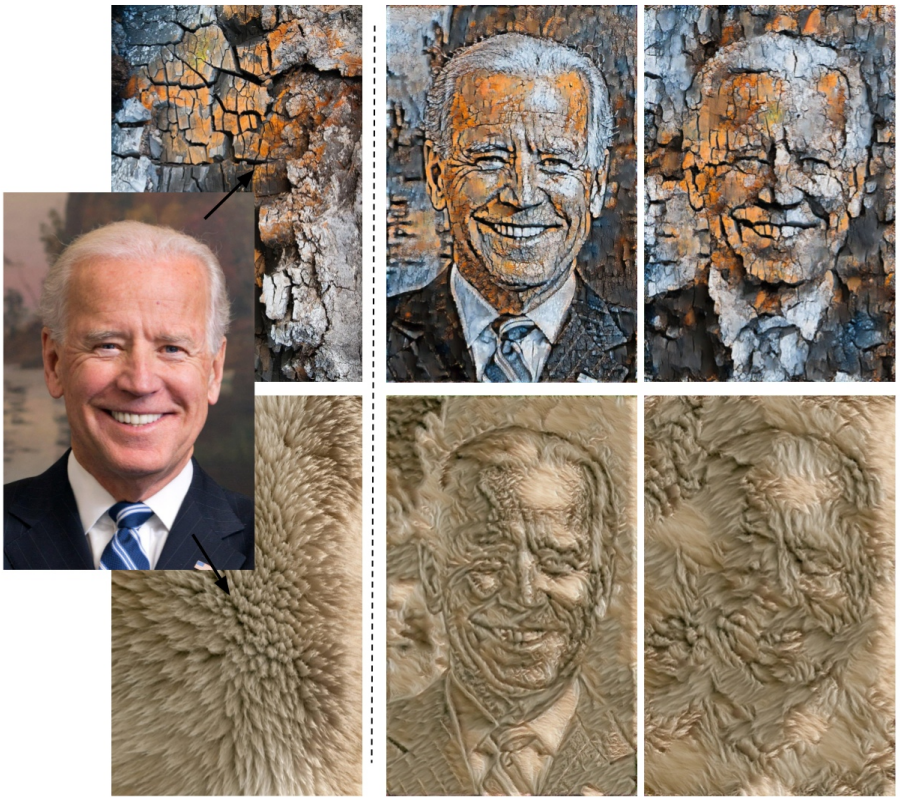

Fig. 6. Image style transfer of an HQ image into two different styles/textures and with two different values for the parameter coarse_dim. The right-most column shows results when using multiple pyramid level (specifically coarse_dim=512) and the second column from the right is showing the results with a single pyramid level (coarse_dim=1024).

3.3 Image style transfer

As described in the paper the way we perform style transfer is by starting the optimization process from a content image and use the style image as the target. We found that the best outputs are generated using a single pyramid level and using a patch size of 11. Of course the best parameters may change from image to image. Figure 6 shows generation of a high resolution image of size 1024x642 with two different texture "styles". We also show the output when using multiple pyramid level for reference. These are the parameters we used for style transfer:
pyramid_factor=1, coarse_dim=max_dim, scale_factors=(1,1), init_mode=content_image, noise_sigma=0, patch_size=11, stride=1, num_projections=64, learning_rate=0.05, num_optimization_steps=300

3.4 Image texture synthesis

The examples in the papers together with the results on figures 7 are all generated with with starts from noise and with **scale_factors=(2,2)** that is:

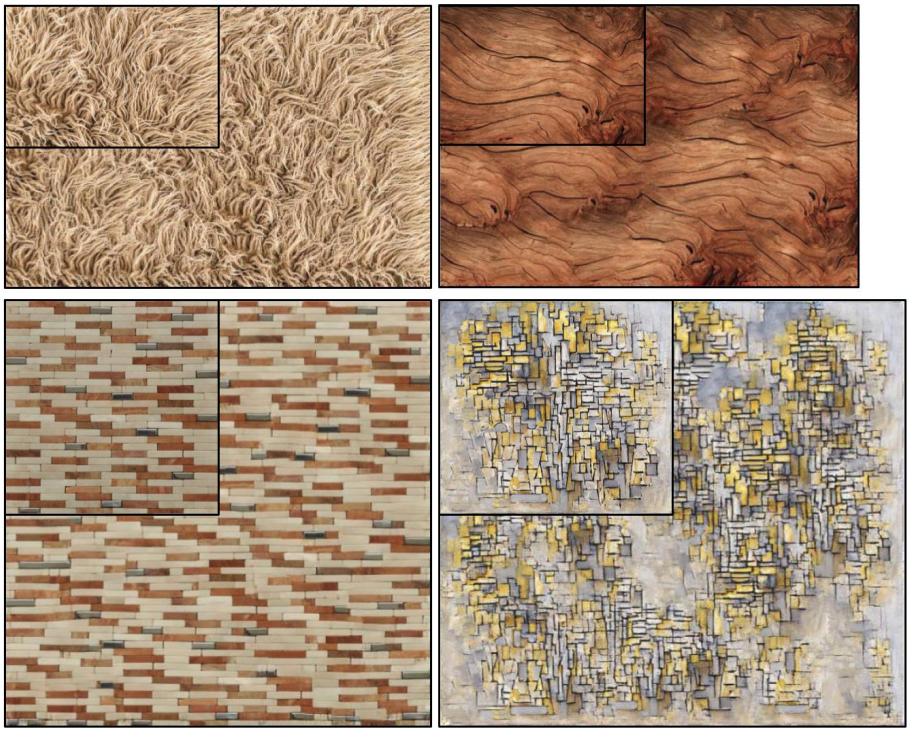

Fig. 7. Results of GPDM on texture synthesis creating a twice as big a texture from a given sample that shows in the upper left cornet of each image.

init_mode='zeros', noise_sigma=1.5, patch_size=7, stride=1, scale_factors=(2,2), num_projections=64, learning_rate=0.05
num_optimization_steps=300

4 GPNN with Approximate nearest neighbor

4.1 The α parameter

In the paper we discussed the possibility to use approximate nearest neighbor (ANN) methods in order to increase GPNN's speed. The main problem with this approach is that it is much harder to control the completeness of the output with such methods. As mentioned in the text, GPNN approximately optimizes a bidirectional loss by adding a term to the patch distance that penalizes patches that have already been used. Specifically, GPNN chooses a patch that minimizes:

$$S_{ij} = \frac{D_{ij}}{\alpha + \min_l D_{lj}}$$

As written in the GPNN paper " The parameter α is used as a knob to control the degree of completeness, where small α encourages completeness, and $\alpha >> 1$ is essentially the same as using MSE.".

α is set to 0.005 in the standard GPNN (the penalty is harder for smaller values of α). When using approximate nearest neighbor we just use MSE (this is equivalent to $\alpha >> 1$ which we mark as "no-α").

Here we add more results from the experiment described in figure 7 of the paper showing the effect of using GPNN with approximate nearest neighbor calculations. Figure 8 shows the effect of using ANN in the reshuffling task. The optimization process in this figure starts from a blurred version of the target (init-mode='blurred_target') which makes exact-NN more prone to selecting smooth patches. The figure shows the crucial effect of the α parameter on the completeness of the outputs but also that using ANN leads to very similar result to exact-NN when α is not used.

In Figure 9 we show the same effect in the style transfer task with GPNN. Notice how using α makes the result contain more details from the style image. Moreover using ANN does not seem to have a negative effect compared to exact-NN without the α parameter.

4.2 Accuracy of the inverted index search

In the main text we claimed that the drop in the results quality when using approximate nearest neighbor is due to not being able to set α rather then to the approximate search being inaccurate. We show here a visual and quantitative evidence to support this claim.

In order to test the accuracy of the inverted axis ANN approach we randomly split the patches of a natural image into two sets and for each patch in the first

Table 1. Comparing the accuracy of two approximate nearest neighbor search methods

	Recall top-1 pick	Recall top-10 pick	distance overhead
FaissIVF	0.76 ± 0.07	0.77 ± 0.07	$133\% \pm 0.11\%$
FaissIVF-PQ	0.64 ± 0.14	0.76 ± 0.07	$135\% \pm 0.09\%$

set searched its nearest neighbor in the other. We compared the approximate results to the exact results. Table 1 shows that FaissIVF was Able to retrieve the same Nearest neighbor patch in $73\% \pm 0.72$ of the patches (Recall top-1 pick). Surprisingly, looking at the 10 best picks of IVF for each patch didn't improve the results much (Recall top-10 pick). The last column in the table (Distance overhead) shows the average quotient of distance to the Approximate nearest neighbor over the distances to the exact nearest neighbor averaged over patches where the first pick differ. These statistics show that in the cases where IVF picked up the wrong patches it picked patches that are 33% more distant then the real nearest neighbor.

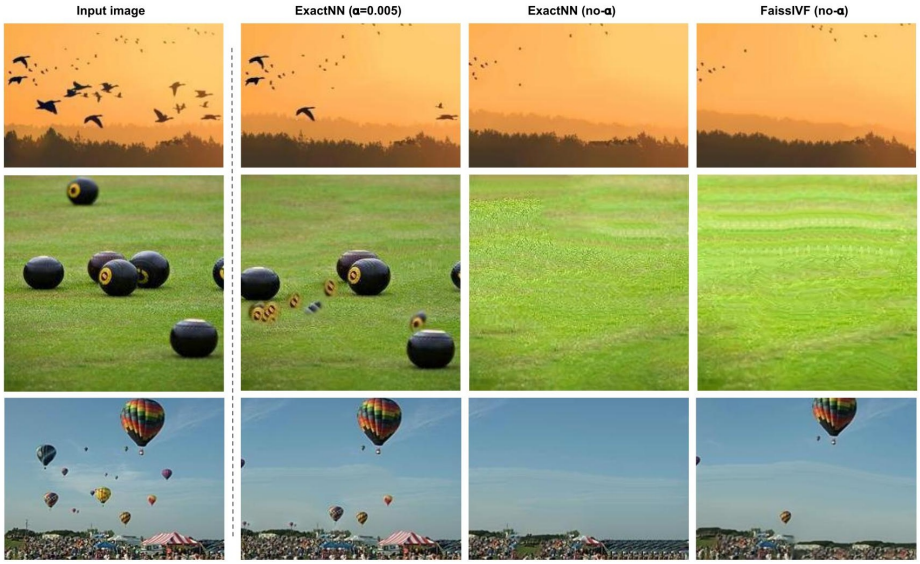

Fig. 8. GPNN reushuffling results on images from the SIGD dataset with different NN computation methods.

Figures 8 and 9 correspond with Figure 7 in the paper. For image reshuffling and style-transfer we compare the results of GPNN with exact nearest neighbor to the results when using IVF only here we also show the results when using exact-nearest neighbor without the effect of the α parameter (no-α). This way we can isolate the effects of choosing approximate nearest neighbor methods from that of the α parameter. As can be seen in these figures, Turning α off drastically lowers the quality of the results even when using exact nearest neighbor. Moreover when not using α the results with approximate and exact nearest neighbor

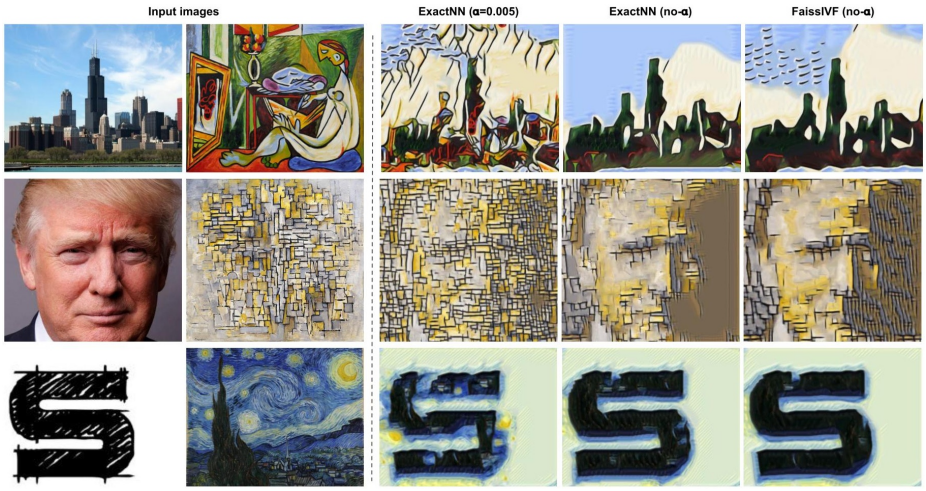

Fig. 9. GPNN style transfer results on images from the SIGD dataset with different NN computation methods. The input content and style images are on the left of each row.

search are comparable. Both show large smooth areas that are presumably due to repeated use of the same patches.

5 Run-time quality trade-off

As mentioned in the paper, decreasing the number of random projections can speed up our algorithm considerably but at a price of image quality. Figure 10 shows this tradeoff for a particular input image.

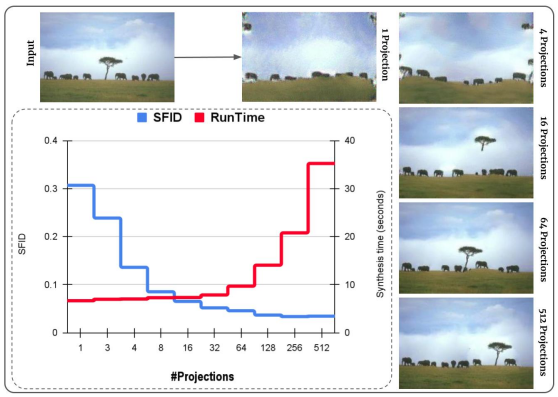

Fig. 10. Average SIFID and compute time of image-reshuffling on SIGD16. The images show generations of one of the images.