Supplementary Material: Program-Guided Image Manipulators

The supplementary material is organized as follows. In Section A, we first discuss the detailed implementation of PG-IM. We then, in Section B, provide ablation studies evaluating the effects of PG-IM's different modules or designs. Finally, we show more results produced by PG-IM in Section C, and also quantitatively compare them with results by the baselines.

A. Implementation Details

We discuss the implementation details necessary to reproduce our results. Our code and data are available on the project page: http://pgim.csail.mit.edu/.

A.1. Repeated Object Detection

We replicated the algorithm developed by Lettry et al. [4]. The algorithm works on nearly-regular images and detects the repeated objects within the image. The model builds on the assumption that features learned from ImageNet [1] encode visual concepts at various levels: from edges to patches to high-level semantic objects. The algorithm takes as input the feature map from a pretrained AlexNet [3] and outputs a set of 2D coordinates representing the centroids of repeated objects. In this section, we explain the algorithm as six-step procedure.

1) For each input feature map $\mathbf{f}$, it extracts all activated neurons at different locations. Denote the neuron at the 2D position (i, j) being activated by $act(\mathbf{f}_{i,j}) = 1$, which is given by

$$act(\mathbf{f}_{i,j}) = \mathbf{1}\left[\mathbf{f}_{i,j} == \max_{(u,v),|u-i|\leq K,|v-j|\leq K} \mathbf{f}_{u,v}\right],$$

where $\mathbf{1}[\cdot]$ is the indicator function, and K is a hyperparameter for "kernel size." Feature maps at different layers of the AlexNet use different kernel sizes.

2) All activated 2D neurons are collected. For each pair of activated neurons from the same feature map, a displacement vector is computed. For simplicity, we use $d^{i,j} = (d_x^{i,j}, d_y^{i,j})$ to index the j-*th* displacement vector from the i-*th* feature map. A majority voting is performed over all displacement vectors. The output of this step is an image-level displacement vector $d^* = (d_x^*, d_y^*)$ between centroids of different objects

$$d_x^* = \arg\max_x \sum_{i,j} \exp(-\|d_x^{i,j} - x\|_2^2) \qquad \text{and} \qquad d_y^* = \arg\max_y \sum_{i,j} \exp(-\|d_y^{i,j} - y\|_2^2).$$

3) Based on the image-level displacement vector d^*, all displacement vectors are filtered. A vector is selected if

$$\|d^{i,j} - d^*\|_2 \leq 3 \times \alpha,$$

where α is a scalar hyperparameter of radius.

4) Each feature map is weighted by the number of selected displacement vectors from this feature map. Only highly weighted feature maps are selected.

5) For each 2D coordinate (x, y), we define a helper function

$$\mathcal{M}(x, y) \triangleq (x \bmod d_x^*, y \bmod d_y^*).$$

An offset o^* is then determined by optimizing the following objective:

$$o^* = \arg\min_o \sum_{i,j} \left(w_{i,j} \|M(d^{i,j} - o) - \frac{d^*}{2}\| \right),$$

where

$$w_{i,j} = \frac{1}{K_i + \phi} \cdot \exp\left(-\frac{\|d^{i,j} - d^*\|_2^2)}{2\alpha^2}\right).$$

K_i is the number of displacement vectors of feature map i. ϕ and α are hyperparameters. In our implementation, we use gradient descent to find the optimal o^*.

6) For each object region detected, the coordinates for all activated neurons within the region vote for the centroid. This step allows distortions of objects that may cause irregular lattices.

A.2. Adaptive Network Depth for PatchGAN

In order to encourage creative and sharp patch generation, we employ a discriminator loss provided by AdaPatchGAN, our adaptive variant of PatchGAN [2], in addition to the L1 loss. In the original PatchGAN, the classifier architecture is fixed with a receptive field of a certain size. However, in images of our interest, the scale of texture varies violently across different images: the repeated pattern is sometimes a small entity but appearing many times, while other times the pattern is a larger object appearing only a few times. Given a single image, since its inferred program description provides a way of dividing it into patches, AdaPatchGAN computes a rough patch size and automatically adds convolutional layers until its receptive field is large enough to cover this patch size. In practice, we observe that our AdaPatchGAN enables the generator to produce more realistic patches than the original PatchGAN.

A.3. Extrapolation as Recurrent Inpainting

As mentioned in the main text, a single neural painting network (NPN) trained on the task of image inpainting is able to perform additional manipulation tasks including extrapolation without any finetuning. The straightforward way of doing this is treating the pixels to extrapolate as missing, and just inpainting them in one go. This approach of treating extrapolation as "feedforward inpainting" of multiple objects, however, results in distorted objects that fail to look realistic, as shown in an ablation study below. This is because NPNs are better at inpainting a localized patch (what they are trained to do as in inpainting) than inpainting a region spanning multiple objects.

Bearing this key observation in mind, our NPNs cast extrapolation into a task of *recurrent* inpainting, where it inpaints one object at a time, conditioned on the context provided by the previous inpainting. Note that this is only possible because the NPNs are guided by programs of images and therefore know how many objects should be extrapolated as well as where each of them should appear. Implementation-wise, this is achieved by repeatedly filling the current empty patch with the network output, and running network inference on the next empty patch. By this design, an empty patch gets inpainted so that it connects seamlessly to both the previously inpainted patches and the original image boundary.

A.4. Detailed Network Specifications and Training Parameters

We append to the end of this document a commented printout of the generator architecture, where channel numbers of the feature maps, kernel sizes, strides, padding, and more can be found. The training of PG-IM generally follows the procedure described in pix2pix [2], except that we do not decay our learning rate after 100 epochs like pix2pix does. Each of our epochs contains 1,000 training samples (i.e., random crops at different scales of the same image of interest), and the networks usually converge within 150 epochs.

B. Ablation Studies

We report three ablations studies to provide intuitions as to why PG-IM works. Key to our approach is the use of program-like descriptions for images in guiding neural painting networks (NPNs) to perform pixel manipulations. The first study demonstrates why such descriptions facilitate the image manipulation tasks. Next, we justify a crucial design for image extrapolation: extrapolation should be cast into the task of *recurrent* inpainting for NPNs, rather than "feedforward inpainting." Supplemental to Figure 7 of the main paper, the third study emphasizes the advantages of considering high-level attributes in the lower-level image manipulation tasks.

B.1. With *vs*. Without Source Patches

Our program-like description for an image informs NPNs of where the pattern of interest (repeatedly) appears in the image. NPNs can then align the available patterns to the corrupted or missing one in order to inpaint it (for simplicity, we discuss only image inpainting in this study, but the conclusion stands for the other tasks). These aligned image patches serve as "source patches," from which NPNs smartly copy to fill in the missing pixels.

Without program-like descriptions, NPNs would not have access to these source patches, which in turn leads to inferior performance such as results with the correct background but missing objects. As Figure 1 shows, when programs (and hence source patches) are not available to the networks to smartly copy from, they fail to ensure the presence of the missing object in their inpainting (e.g., the cross in the left pane and the diamond in the right pane). In contrast, our NPNs exploit the source patches provided by programs and can inpaint the missing objects even if they are corrupted completely in the input. This result supports that PG-IM outperforms repetition-aware but "programless" CNNs.

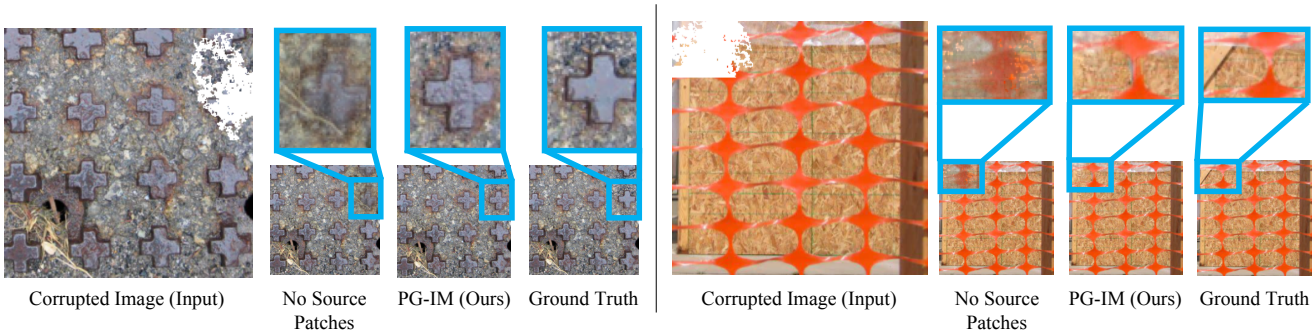

Figure 1: Inpainting with *vs*. without the source patches provided by program. For each pane, the leftmost figure is the corrupted input to inpaint; to the right are results inpainted with and without source patches as well as the ground truth. With programs describing where the object centers are, our NPNs align the intact objects to the corrupted object and exploit such alignment to inpaint the missing pixels. Without such source patches, the networks have difficulty ensuring the "objectness" of the inpainted patch.

B.2. With *vs*. Without Recurrent Inpainting

In this study, we show empirical evidence that NPNs produce better results when extrapolation is cast into the task of recurrent inpainting than feedforward inpainting. As Figure 2 shows, when the entire stripe gets extrapolated in one go ("extrapolation as feedforward inpainting"), the extrapolated objects tend to be distorted, since our NPNs are trained only on more localized inpainting. By treating extrapolation as *recurrent* inpainting, our NPNs are essentially doing, repeatedly, what they are trained on and hence generate more realistic-looking results.

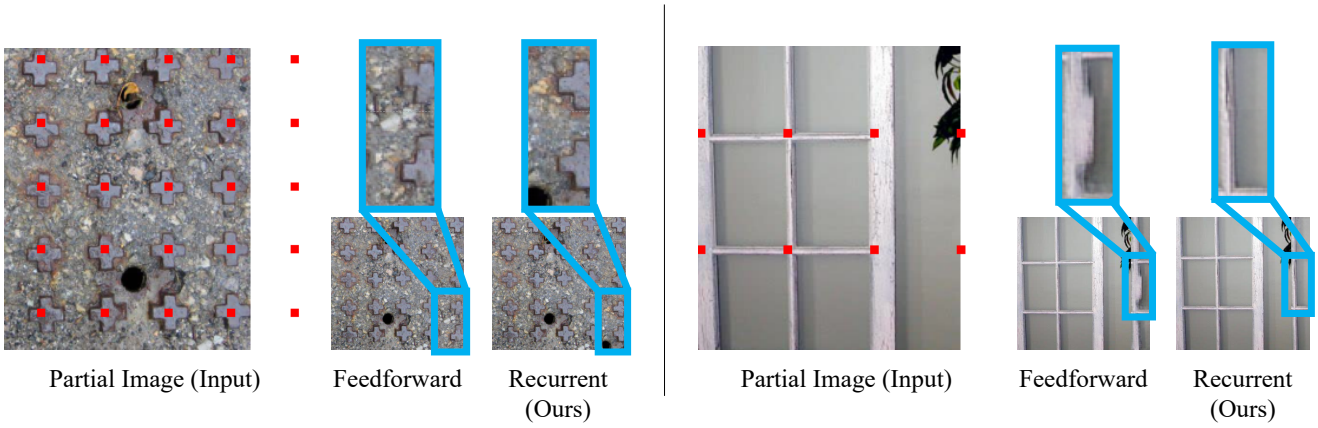

Figure 2: Extrapolation as feedforward *vs*. recurrent inpainting. For each pane, the leftmost figure is the partial input to be extrapolated; to the right are results extrapolated in one go and recurrently. Recall that the networks are trained only on inpainting one entity, so the networks tend to produce distorted objects if they are asked to inpaint the whole stripe spanning multiple entities. Instead, our NPNs treat extrapolation as recurrent inpainting, where they repeatedly solve the task they are good at and therefore produce better results.

B.3. With *vs*. Without Considering Attributes

In Figure 7 of the main paper, we show our program-like descriptions of images can naturally incorporate attributes, enabling the NPNs to perform attribute-aware image manipulation, e.g., by referencing only patches with the same attributes as the patch to inpaint. We supplement two more examples demonstrating why attribute-aware image manipulations are important, and how PG-IM achieves superior performance on such tasks.

As shown in Figure 3 left, the tomato to be inpainted is next to four orange tomatoes and one green tomato. Since convolutions are more of a local operation, one can expect a convolutional generator to produce a tomato with mixed colors of both orange and green, as is the case for PG-IM w/o Attributes. However, we, as humans, reason at a higher level over the entire image, coming to the conclusions that the tomato to inpaint on the left should be orange, and that the candy to inpaint on the right should be red. With similar capabilities, PG-IM correctly preserves the global regularity of attributes in its inpainting.

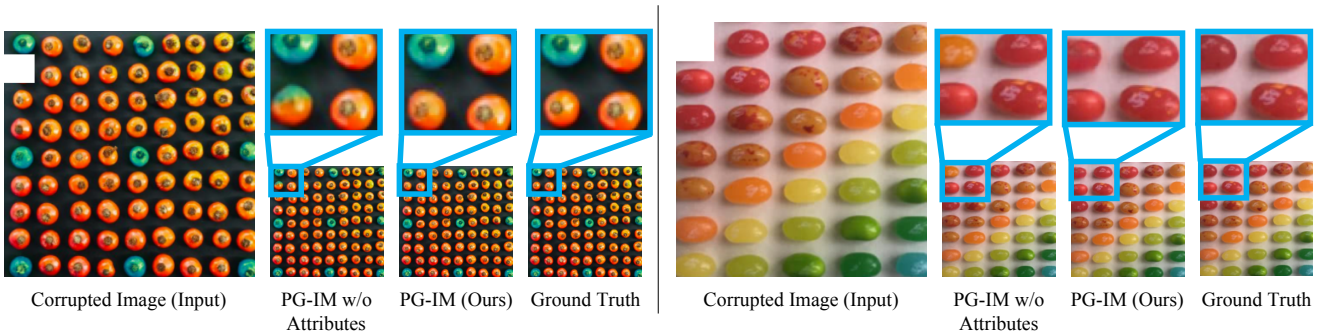

Figure 3: Inpainting with *vs*. without considering object attributes. For each pane, the leftmost figure is the corrupted input to be inpainted; to the right are results inpainted with and without considering attributes as well as the ground truth. The attribute-agnostic networks tend to produce patches with mixed colors that break the global regularity of attributes. PG-IM is able to respect such regularity by referencing only patches with the same attribute as the patch to inpaint.

C. More Experiments and Results

We supplement more results by PG-IM on the tasks of image inpainting, extrapolation, and regularity editing. For PatchMatch and Image Quilting, we search for one set of optimal hyperparameters and use that for the entire test set. Similarly, we train Non-Stationary [6] and PG-IM with single sets of optimal hyperparameters on each test image. For GatedConv [5], we use the trained model released by the authors.

C.1. Inpainting

Figure 4 shows how PG-IM compares in image inpainting with the baselines on another six test images. PG-IM is able to ensure the presence of the repeated object in its inpainting (e.g., the cross structure in Image 1 and the yellow tag in Image 5), while the baselines tend to have the object missing (e.g., PatchMatch completely misses the yellow tag in Image 5) or incomplete (e.g., the pig in Image 3) due to lack of the concept of objects.

C.2. Extrapolation

We supplement four more examples of image extrapolation by PG-IM and how they compare with results by the baselines. As Figure 5 shows, extrapolation by PG-IM is sharp, respects the global regularity (e.g., the "pig array" in Image 3), and seamlessly connects to the original images. In contrast, PatchMatch tends to blur over the image boundaries for smooth transition to the extrapolated contents (e.g., the blurriness over the boundary in Image 2). Image Quilting and GatedConv tend to break the global regularity (e.g. Images 2 and 4). Finally, Non-Stationary should be considered as a super-resolution method instead of an extrapolation one, because it essentially replicates the patterns on a larger canvas and interpolates in between; notice how it does *not* extrapolate beyond the texture boundaries in Images 2 and 4.

C.3. Regularity Editing

We supplement four more examples of image regularity editing by PG-IM in Figure 6. Although PG-IM has been trained only on inpainting, with the program descriptions of images, it is naturally capable of exaggerating the irregularity in the images' global structures.

C.4. Failure Cases

Figure 7 demonstrates a failure case for inpainting. In this challenging case, the transmission and reflection effects make the repetitive patterns less homogeneous, leading to non-photorealistic inpainting.

C.5. Baseline Finetuning

PG-IM learns from a single input image by exploiting the repetition structure, whereas other learning-based methods (GatedConv and PartialConv) learn from a large dataset (Places365). We supplement extra experiments, where we finetune the GatedConv model (pre-trained on Places365) on 50 Facade images and evaluate it on the held-out images (Figure 8). Learning from similar facade images does help the model produce more photorealistic results (Inception score: 1.187 $\rightarrow$

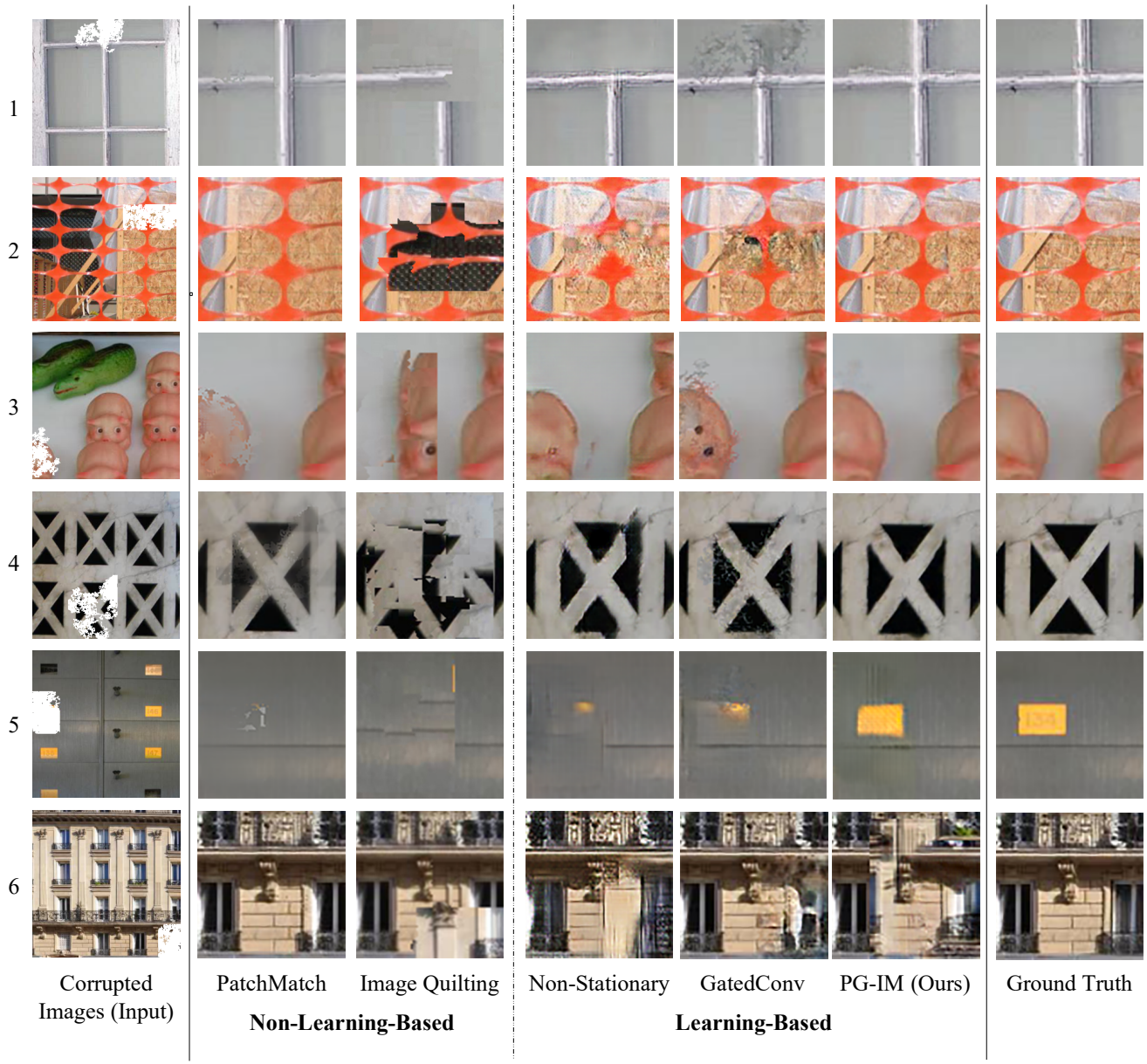

Figure 4: More inpainting results by PG-IM and the baselines. The white pixels in the leftmost column are missing pixels to inpaint; the rightmost column shows the ground-truth patches. For easier comparison, we show only the close-up views of the inpainted region (and its proximity for some context). Aware of the images' global regularity, PG-IM inpaints the missing objects that maintain this regularity. Note this is not necessarily the case for other methods: baselines miss out the pig entity of Image 3 and the yellow tag of Image 5. Although PatchMatch gets the cross structure in Image 1, its result is less realistic than ours. Another advantage of PG-IM over the baselines is the ability to generate pixels that seamlessly connect to the original image contents (compare the results for Images 1, 2, and 4).

1.191). However, the model still fails on structured objects such as windows, which can be well handled by our patch-based, program-guided NPN (PG-IM's Inception score is 1.210).

References

[1] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009. 1

[2] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. In *CVPR*, 2017. 2

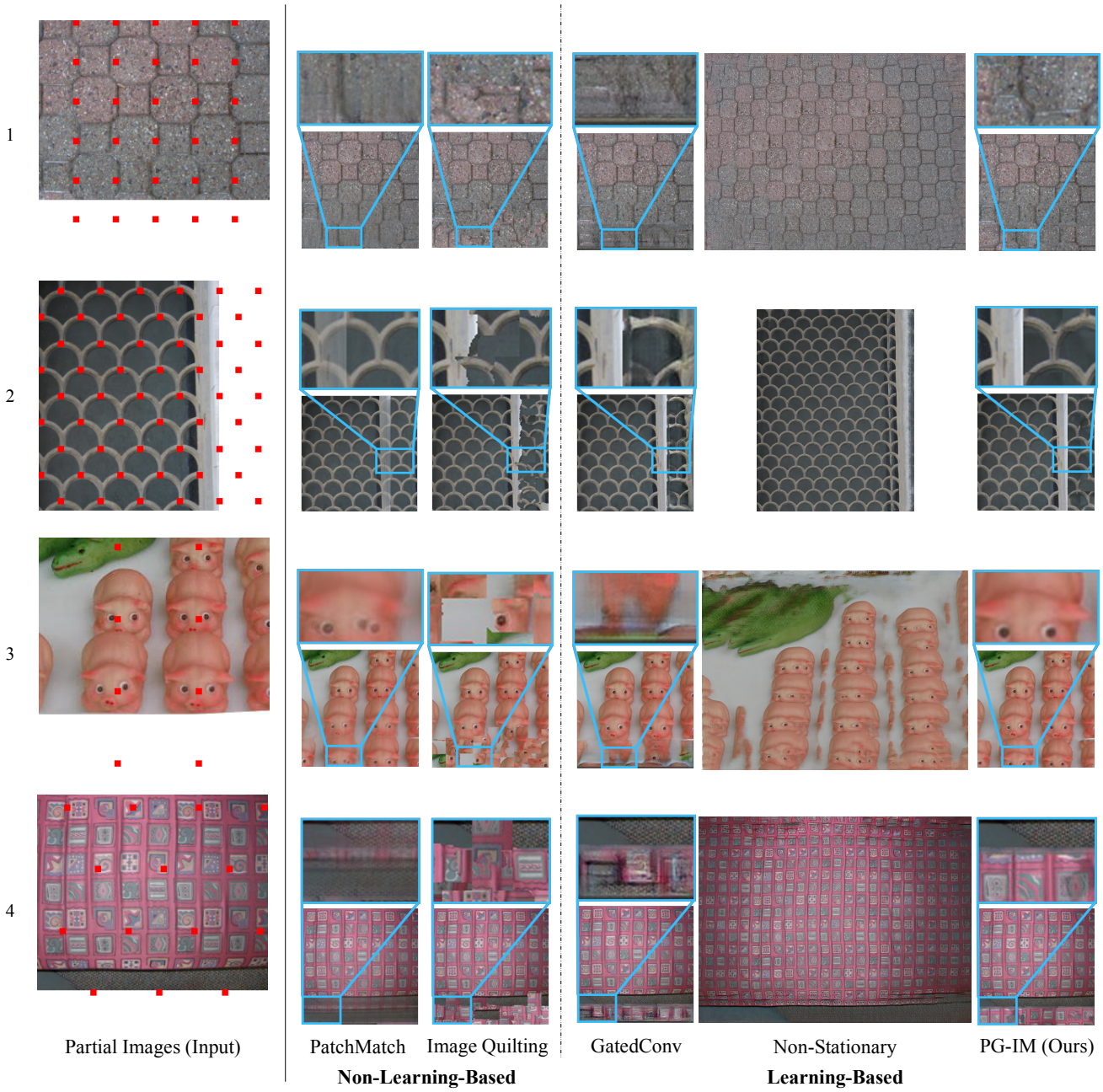

Figure 5: Extrapolation results by PG-IM and the baselines. The white pixels in the leftmost column indicate the pixels to be extrapolated, and the red dots are object centers given by the program descriptions. With such program descriptions, PG-IM knows where to extrapolate to automatically, whereas other methods require the user to specify where to extrapolate to. PG-IM generates realistic images while preserving the global regularity. In contrast, GatedConv fails to capture the regularity; Non-Stationary does not preserve the original content of images; non-learning-based baselines sometimes generate blurry images because of the repetition of similar objects in the partial image.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *NeurIPS*, 2012. 1

[4] Louis Lettry, Michal Perdoch, Kenneth Vanhoey, and Luc Van Gool. Repeated pattern detection using cnn activations. In *WACV*, 2017. 1

[5] Jiahui Yu, Zhe Lin, Jimei Yang, Xiaohui Shen, Xin Lu, and Thomas S Huang. Free-form image inpainting with gated convolution. In *ICCV*, 2019. 4

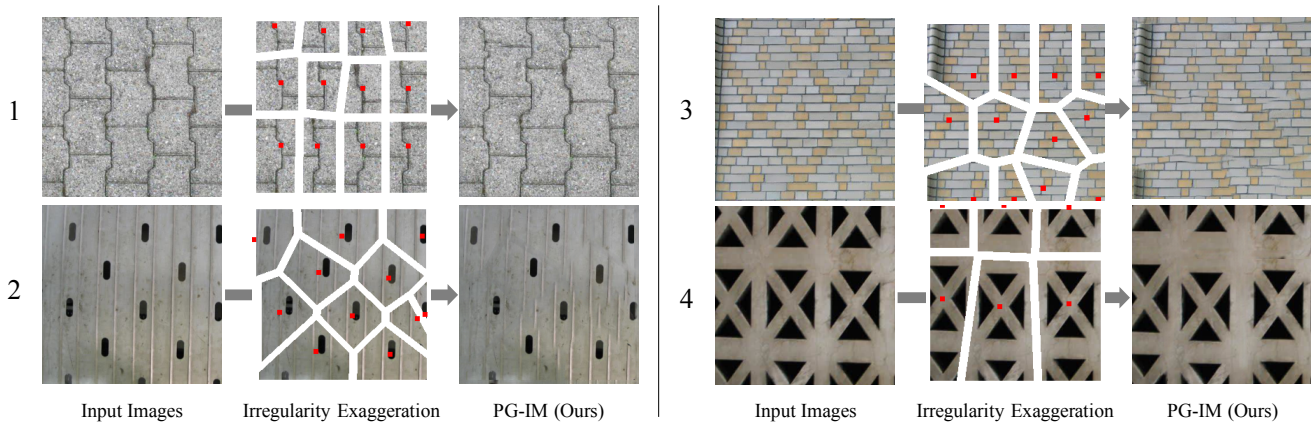

Figure 6: PG-IM enables automated and semantic-aware irregularity exaggeration. By comparing the centroids of the detected objects and the ones reconstructed by the program, we can measure and exaggerate the structural irregularity of input images. In these examples, we first multiply by 2 the displacement vectors between the object centroids provided by the programs and the detected object centroids, and then randomly flip the sign for each displacement vector. According to these new displacement vectors, we shift the patches and then let the NPN fill in the missing pixels.

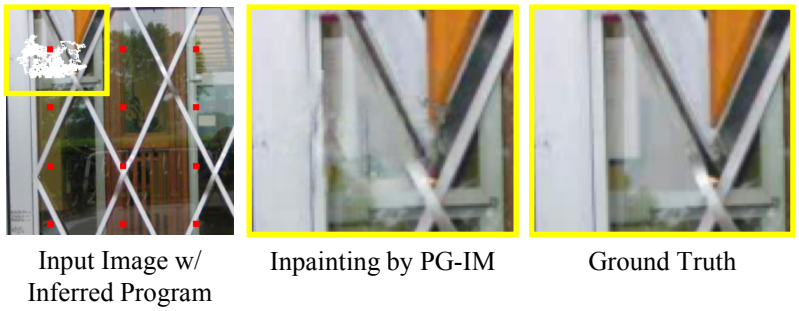

Figure 7: A failure case.

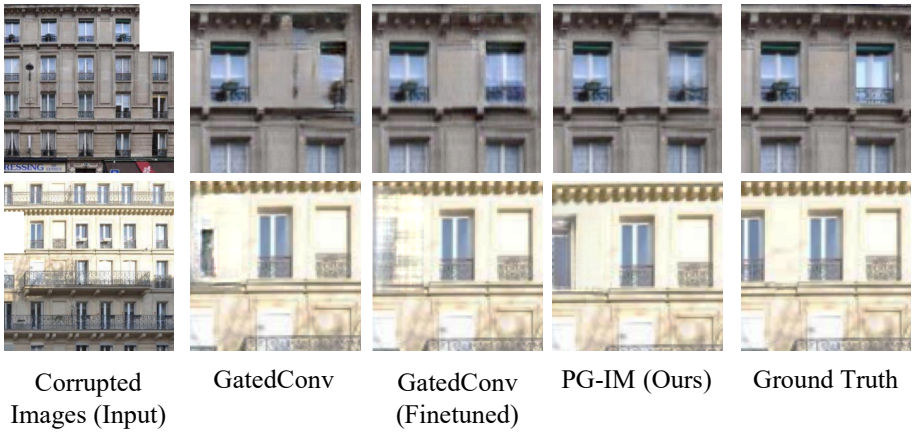

Figure 8: GatedConv model finetuned on the Facade dataset. Note that our PG-IM is only trained on a single input image.

[6] Yang Zhou, Zhen Zhu, Xiang Bai, Dani Lischinski, Daniel Cohen-Or, and Hui Huang. Non-stationary texture synthesis by adversarial expansion. *SIGGRAPH*, 37(4), 2018. 4

Appendix: Detailed Generator Architecture

```
('1', ReppadConv2d(3, 64, 3, stride=1, padding=1)),
('1a', nn.ELU()),
# 2: maxpool across tracks
# 3: concat 2
('4:+pool', ReppadConv2d(128, 96, 1, stride=1)),
('4a', nn.ELU()),
('5', ReppadConv2d(96, 96, 4, stride=2, padding=1)),
('5a', nn.ELU()),
# 6: maxpool across tracks
# 7: concat 6
('8:+pool', ReppadConv2d(192, 128, 1, stride=1)),
('8a', nn.ELU()),
('9', ReppadConv2d(128, 128, 4, stride=2, padding=1)),
('9a', nn.ELU()),
# 10: maxpool across tracks
# 11: concat 10
('12:+pool', ReppadConv2d(256, 256, 1, stride=1)),
('12a', nn.ELU()),
('13', ReppadConv2d(256, 256, 4, stride=2, padding=1)),
('13a', nn.ELU()),
# 14: maxpool across tracks
# 15: concat 14
('16:+pool', ReppadConv2d(512, 384, 1, stride=1)),
('16a', nn.ELU()),
('17', ReppadConv2d(384, 384, 4, stride=2, padding=1)),
('17a', nn.ELU()),
('18', ReppadConv2d(384, 384, 3, stride=1, padding=1)),
('18a', nn.ELU()),
# ----------------
('19', nn.ConvTranspose2d(384, 384, 4, stride=2, padding=1)),
('19a', nn.ELU()),
# 20: maxpool across tracks
# 21: concat 16a, 20
('22:+pool+16a', ReppadConv2d(1152, 384, 1, stride=1)),
('22a', nn.ELU()),
('23', ReppadConv2d(384, 384, 3, stride=1, padding=1)),
('23a', nn.ELU()),
('24', nn.ConvTranspose2d(384, 256, 4, stride=2, padding=1)),
('24a', nn.ELU()),
# 25: maxpool across tracks
# 26: concat 12a, 25
('27:+pool+12a', ReppadConv2d(768, 256, 1, stride=1)),
('27a', nn.ELU()),
('28', ReppadConv2d(256, 256, 3, stride=1, padding=1)),
('28a', nn.ELU()),
('29', nn.ConvTranspose2d(256, 192, 4, stride=2, padding=1)),
('29a', nn.ELU()),
# 30: maxpool across tracks
# 31: concat 8a, 30
('32:+pool+8a', ReppadConv2d(512, 192, 1, stride=1)),
('32a', nn.ELU()),
('33', ReppadConv2d(192, 192, 3, stride=1, padding=1)),
('33a', nn.ELU()),
('34', nn.ConvTranspose2d(192, 96, 4, stride=2, padding=1)),
('34a', nn.ELU()),
# 35: maxpool across tracks
```

```
# 36: concat 1a, 35
('37:+pool+1a', ReppadConv2d(256, 96, 1, stride=1)),
('37a', nn.ELU()),
('38', ReppadConv2d(96, 96, 3, stride=1, padding=1)),
('38a', nn.ELU()),
# 39: maxpool across tracks
('40:pool', ReppadConv2d(96, 64, 3, stride=1, padding=1)),
('40a', nn.ELU()),
('41', ReppadConv2d(64, nch_out, 3, stride=1, padding=1)),
```