GAUSSIFIER: FAST 2D GAUSSIAN DECOMPOSITION

Anonymous authors
Paper under double-blind review

ABSTRACT

2D Gaussians render images at thousands of frames per second. However, fitting an image takes seconds of optimization. This confines 2D Gaussians to offline use. Feedforward methods are faster but less accurate without per-image fine-tuning. We present Gaussifier, a lightweight feedforward framework for 2D Gaussian decomposition. We learn a density map of Gaussian centers from per-image fitting data. Equal-mass Voronoi places Gaussians so that each Voronoi cell holds approximately equal density. A stateless, reusable network then refines the output in a few iterations. Gaussifier reaches 40.73/38.61 dB PSNR on Kodak-24/DIV2K at over 60 FPS. At the same Gaussian count, it exceeds every feedforward baseline by 3.0 to 15.6 dB in PSNR. With fewer Gaussians than any baseline, it still leads the strongest one by 2.5 to 3.5 dB. At the same count, baselines with their own per-image fitting need 26 to 326 times longer to reach its PSNR. Gaussifier makes high-fidelity 2D Gaussian images fast and light enough for real-time applications.

1 INTRODUCTION

Gaussian images represent an image as a set of explicit 2D Gaussian primitives, each with its own position, shape, and color (Zhang et al., 2024b; 2025; Li et al., 2026). These sparse representations are increasingly used in many applications, including vision-language models (Omri et al., 2026), image restoration (Chen et al., 2026d), and video coding (Wang et al., 2025). However, fitting Gaussians to each image takes seconds of optimization. Existing feedforward models rely on per-image fine-tuning or give up quality (Zeng et al., 2025; Wang et al., 2026; Zeng et al., 2026). So far, high-fidelity 2D Gaussian representations remain limited to offline use because of this bottleneck.

An ideal decomposition reconstructs the input faithfully with few primitives and runs fast. Existing methods trade these goals against one another. For example, a codec trades quality for size. Furthermore, decomposing an image into Gaussians is an ill-posed problem. Many different sets of Gaussians can render nearly the same image. The decomposition must jointly decide the Gaussian count, locations, shapes, and colors. Since overlapping Gaussians can contribute to the same pixel, the best parameters for one Gaussian depend on its neighbors. Fitting resolves these dependencies with thousands of gradient steps. Feedforward methods still rely on this fitting for final polishing.

We present Gaussifier, a lightweight feedforward framework. It learns a density map of Gaussian centers from per-image fits, places Gaussians on it, and refines them with a few learned updates. At the same count, it exceeds every feedforward baseline in peak signal-to-noise ratio (PSNR) by 3.0 to 15.6 dB, and baselines need orders of magnitude more fitting time to match its PSNR. At its predicted count, it leads every baseline by over 8 dB. We make three contributions:

- **Learned sparse allocation.** A lightweight model predicts the expected number of Gaussian centers at each pixel. The sum of this density map sets the Gaussian count, which users can scale up or down. Equal-mass Voronoi (EMV) sampling places the Gaussians so that each Voronoi cell holds approximately equal density mass. The resulting layout resembles that of the fitted Gaussians.
- **Stateless refinement without fitting gradients.** One refinement network jointly updates all Gaussians from the rendering residual. Six updates add about 15 to 16 dB. The same weights serve any Gaussian count, and one to six updates trade speed for quality.
- **Video-rate and high-fidelity decomposition.** Gaussifier decomposes a 512×512 image in under 15 ms on average and reaches 40.73/38.61 dB PSNR on Kodak-24/DIV2K. On Kodak-24, at the same Gaussian count, it matches the PSNR achieved by GaussianImage with 20 s of fitting. This moves 2D Gaussian images from offline to real-time and large-scale use.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

2 Related Work

Optimization-based Gaussian image representations. Implicit neural representations fit a network to each image and evaluate it at every pixel (Sitzmann et al., 2020; Müller et al., 2022; Dupont et al., 2021). GaussianImage (Zhang et al., 2024b) brings 3D Gaussian Splatting (3DGS) (Kerbl et al., 2023) to images and renders a fitted image at over 1,000 frames per second (FPS). Image-GS (Zhang et al., 2025), GaussianImage++ (Li et al., 2026), and structure-guided allocation (Liang et al., 2025) improve the placement of the Gaussians with error-guided growth, densification, and structural priors. All of these methods fit each image by iterative optimization, which takes seconds per image.

Feedforward Gaussian images. Feedforward approaches predict Gaussian parameters with a trained network instead of per-image optimization. Instant-GI dithers a predicted probability map of Gaussian positions into points (Zeng et al., 2025). Capacity-constrained point distributions instead give every point an equal share of the density (Balzer et al., 2009; De Goes et al., 2012). Fast-2DGS samples Gaussian positions from a spatial prior conditioned on the Gaussian count. A separate network then predicts dense Gaussian attributes (Wang et al., 2026). Instant-GI and Fast-2DGS both need per-image fine-tuning to reach high fidelity. AIR avoids optimization by adding Gaussians stage by stage from the reconstruction residual (Zeng et al., 2026). However, earlier Gaussians stay fixed, so later stages cannot revise them. Feedforward 3DGS predicts Gaussians anchored to the pixels of one or a few input images (Szymanowicz et al., 2024; Charatan et al., 2024; Chen et al., 2024; Zhang et al., 2024a). ReSplat and GIFSplat can refine these Gaussians iteratively from rendering errors without test-time gradients (Xu et al., 2026; Chen et al., 2026a).

Gaussian image applications. Gaussian primitives are compact, editable, and fast to render. Gaussian tokenizers (Dong et al., 2025; Zhang et al., 2026) and super-resolution networks (Hu et al., 2025; Chen et al., 2025) predict Gaussians inside a trained model. Other methods decompose the input image. GViT classifies images from a few hundred Gaussians optimized jointly with the classifier (Hernandez et al., 2025). GaussianVision aligns Gaussian images with language after fitting 12.8M images (Omri et al., 2026). Zero-shot restoration fits a low-light or hazy image for 20-30k iterations and adjusts its Gaussians (Chen et al., 2026d;b;c). MiraGe edits images through flat Gaussians fitted to the image (Waczyńska et al., 2024). Gaussian codecs transmit quantized fitted Gaussians (Zhang et al., 2024b; Wang et al., 2025). GSVC decodes at over 1,500 FPS but takes about 200 s to encode each 1080p frame (Wang et al., 2025). In these decomposition methods, producing the Gaussians is a major bottleneck because each image or frame needs its own optimization.

3 Method

3.1 Gaussian Representation and Method Overview

Given an image I on a pixel grid Ω with P pixels, we can construct N anisotropic 2D Gaussians. Each Gaussian has a continuous position $\boldsymbol{\mu}_i$, a covariance $\boldsymbol{\Sigma}_i$, and a color coefficient $\boldsymbol{a}_i$. Following additive Gaussian image representations (Zhang et al., 2024b), we reconstruct the image as

$$\widehat{I}(\boldsymbol{x}) = \mathrm{clip}_{[0,1]}\left[\sum_{i=1}^{N} \boldsymbol{a}_i \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_i)^\top \boldsymbol{\Sigma}_i^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_i)\right)\right]. \tag{1}$$

We parameterize the covariance by its matrix logarithm $\boldsymbol{S}_i = \log \boldsymbol{\Sigma}_i$ (Arsigny et al., 2006). The covariance $\boldsymbol{\Sigma}_i = \mathrm{expm}(\boldsymbol{S}_i)$ then stays positive definite under additive updates. For rendering, we convert $\boldsymbol{S}_i$ to the equivalent principal scale and rotation. Based on this representation, our framework is illustrated in Figure 1.

3.2 Gaussian Sampling

Design. Gradient-based optimization produces well-placed Gaussians in both 2D and 3D splatting (Zhang et al., 2024b; Kerbl et al., 2023). Feedforward models trained only with a photometric loss often place Gaussians in repeating patterns that do not follow the image content. This degrades their visual quality. We therefore use Gaussians fitted by per-image optimization as teachers for learned sampling. We first predict a Gaussian density map from the input image. A sampler then places Gaussians according to this map. Their layout resembles that of the optimized 2D Gaussians.

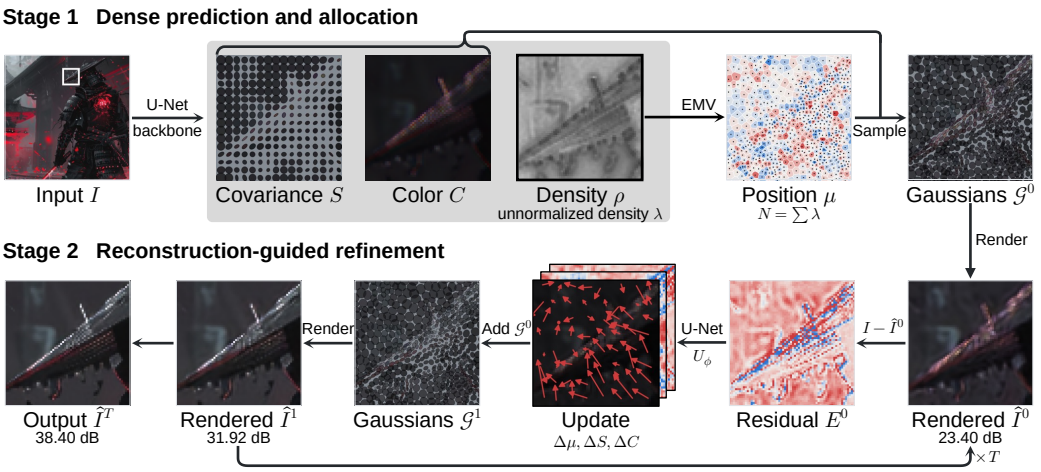

Figure 1: **Overview of Gaussifier.** Panels after the input show the region in its white box. In Stage 1, a U-Net maps the image I to log-covariance, color, and density maps S, C, and λ. The sum of λ gives the Gaussian count N. EMV places N centers $\boldsymbol{\mu}$ from the normalized density ρ. Sampling S and C at these centers gives the initial Gaussians $\mathcal{G}^0$. In Stage 2, a stateless refinement network U_ϕ reads the image, the Stage-1 maps, the current rendering $\widehat{I}^t$, its residual E^t, and the count map H^t. For simplicity, the diagram shows only the residual input. It predicts updates $\Delta\boldsymbol{\mu}$, $\Delta\boldsymbol{S}$, and $\Delta\boldsymbol{C}$ to the center, log-covariance, and color of every Gaussian, while N stays fixed. This repeats T times. Displacement arrows in $\Delta\boldsymbol{\mu}$ are magnified for visibility.

Density map. For efficiency, a 2M-parameter U-Net (Ronneberger et al., 2015) predicts the expected number of Gaussian centers in each pixel, $\lambda(\boldsymbol{x})$. It also predicts the initial log-covariance $S(\boldsymbol{x})$ and color $C(\boldsymbol{x})$ of a Gaussian centered at each pixel $\boldsymbol{x}$. Inspired by the training strategy in bit2bit (Liu et al., 2024), we train λ using sparse rasterized targets that equal 1 at the centers of fitted reference Gaussians and 0 elsewhere. The exact positions of fitted centers vary even between similar image regions. The prediction λ therefore becomes their expected density, which is smooth. This density carries two pieces of information. Its intensity profile tells where the Gaussians should be. Its sum defines a total Gaussian count, $N = \sum_{\boldsymbol{x}\in\Omega} \lambda(\boldsymbol{x})$, as in density-based object counting (Lempitsky & Zisserman, 2010). We keep only the intensity profile by dividing λ by its mean, $\rho(\boldsymbol{x}) = P\,\lambda(\boldsymbol{x})/N$. The relative density ρ averages to one, so $\rho(\boldsymbol{x}) = 2$ marks a region that needs twice the average density of Gaussians. A user can also scale N up or down while keeping the same ρ (Appendix A.3).

Equal-mass Voronoi sampling. EMV turns ρ into N Gaussian centers by treating placement as a coverage problem. We partition the image into N Voronoi cells V_i with one center in each. Every cell should hold approximately the same density mass, $m_i = \sum_{\boldsymbol{x}\in V_i} \rho(\boldsymbol{x}) \approx \bar{m}$, where $\bar{m}$ is the mean cell mass. Highly textured regions receive both more and smaller cells. This is similar to capacity-constrained Voronoi tessellations, which enforce equal mass exactly but require many iterations (Balzer et al., 2009). Random sampling leaves clumps and gaps, while dithering matches the local count but the cells around its centers still hold unequal mass. Lloyd relaxation alone spreads centers in proportion to $\rho^{1/2}$ and under-serves detailed regions (Lloyd, 1982; Du et al., 1999). Inspired by weighted Linde-Buzo-Gray stippling (Deussen et al., 2017), EMV approximates equal mass by oversampling sites from ρ and merging the lightest cells (Algorithm 1 and Appendix A.2). Weighted Linde-Buzo-Gray stippling splits heavy cells and deletes light ones over many iterations. EMV never splits cells. It starts from about $1.5N$ sites and merges light cells into their neighbors until N remain. A few Lloyd-relaxation iterations then even out the local spacing and cell masses. The sites become the initial centers $\boldsymbol{\mu}_i^0$, and the final cells give the masses m_i. Placement takes about 2 to 3 ms for 40k-70k Gaussians on an RTX 5090.

Initial Gaussians. Each Gaussian takes its initial log-covariance and color from S and C at its center by bilinear interpolation $\mathcal{B}$, $\boldsymbol{S}_i^0 = \mathcal{B}(S, \boldsymbol{\mu}_i^0)$ and $\boldsymbol{C}_i^0 = \mathcal{B}(C, \boldsymbol{\mu}_i^0)$. Because EMV chooses

Algorithm 1 Approximate EMV placement in pixel coordinates.

Require: Density ρ on Ω and Gaussian count N
Ensure: Initial centers $\{\boldsymbol{\mu}_i^0\}$ and cells $\{V_i\}$
Sample about $1.5N$ sites $Q = \{\boldsymbol{q}_i\}$ from ρ with stratified sampling
while $|Q| > N$ **do**
    Compute the Voronoi cells $\{V_i\}$ of Q and their masses $m_i = \sum_{\boldsymbol{x}\in V_i} \rho(\boldsymbol{x})$
    Merge the lowest-mass cells into their lowest-mass neighbors
    Move each kept site to the mass-weighted mean of itself and the sites merged into it
end while
for $k = 1, 2, 3$ **do**
    Recompute $\{V_i\}$ and move each nonempty site to $\sum_{\boldsymbol{x}\in V_i} \rho(\boldsymbol{x})\boldsymbol{x} / \sum_{\boldsymbol{x}\in V_i} \rho(\boldsymbol{x})$
end for
Recompute $\{V_i\}$ for the final sites
return $\boldsymbol{\mu}_i^0 = \boldsymbol{q}_i$ and the cells $\{V_i\}$

the centers by discrete sampling and merging, reconstruction gradients cannot reach the density map through the center locations. We therefore scale the color of each Gaussian by its relative cell mass, $w_i = m_i/\bar{m}$. Each m_i sums ρ over its cell, so the reconstruction gradient on w_i reaches every pixel of the cell. This gradient adjusts the shape of the density but not the count (Appendix A.2). During pretraining, a density loss (Eq. 4) also supervises λ directly. The cell-mass weights are computed once and stay fixed while the centers move. However, the refinement network described in the next section still updates the unweighted color $\boldsymbol{C}_i^t$, while Gaussian i uses the color coefficient $\boldsymbol{a}_i = \mathrm{clip}_{[0,1]}(w_i \boldsymbol{C}_i^t)$ in Eq. 1 to render images.

3.3 Reconstruction-Guided Refinement

Design. EMV gives a good initial approximate allocation. However, overlapping Gaussians contribute to the same pixel, so the attributes of each Gaussian depend on its neighbors. Per-image fitting resolves this with thousands of gradient steps. Each step is driven by the rendering loss and is slow. To avoid this, we train a network that reads the errors and updates all Gaussians together, iteratively. The network can learn to coordinate attribute changes from millions of images, so that a few inference iterations can replace about a hundred gradient steps.

Update network. At step t, we render the current Gaussians to obtain $\widehat{I}^t$ and the residual $E^t = I - \widehat{I}^t$. We also rasterize the current centers into a count map H^t, which shows where the Gaussians currently lie. The network U_ϕ reads these maps and the initialization context $Z = [I, \rho, S, C]$. It predicts a dense update field, which we sample at each center to update the Gaussians:

$$D^t = U_\phi\Big([Z, \widehat{I}^t, E^t, H^t]\Big), \qquad (\Delta\boldsymbol{\mu}_i^t, \Delta\boldsymbol{S}_i^t, \Delta\boldsymbol{C}_i^t) = \mathcal{B}(D^t, \boldsymbol{\mu}_i^t), \tag{2}$$

$$\boldsymbol{\mu}_i^{t+1} = \Pi_{\mathrm{img}}(\boldsymbol{\mu}_i^t + \Delta\boldsymbol{\mu}_i^t), \qquad \boldsymbol{S}_i^{t+1} = \boldsymbol{S}_i^t + \Delta\boldsymbol{S}_i^t, \qquad \boldsymbol{C}_i^{t+1} = \mathrm{clip}_{[0,1]}(\boldsymbol{C}_i^t + \Delta\boldsymbol{C}_i^t), \tag{3}$$

where Π_{img} projects centers onto the image domain and $\Delta\boldsymbol{S}_i^t$ is symmetric.

Stateless updates. Our refinement is inspired by ReSplat, which refines 3D Gaussians from rendering errors (Xu et al., 2026). ReSplat keeps a hidden state for every Gaussian and updates it with a transformer that attends over neighbors. Its cost grows with the number of Gaussians. Our network is stateless and carries no hidden state between updates. We use a small U-Net with 0.75M parameters instead of a transformer. It reads and predicts image-sized maps, so the same set of weights works for any number of Gaussians (Appendix A.3), and its cost does not grow with the Gaussian count. All $T = 6$ updates share the same parameters ϕ, and each update costs one pass of the refinement network and one rendering.

3.4 Training Objectives

Pretraining data. We pretrain the 2M-parameter U-Net on Open Images data (Kuznetsova et al., 2020). Each image is paired with reference Gaussians fitted to it by per-image optimization. We first

train on about 107k such pairs. We then fine-tune on about 20k pairs whose fits start from an earlier Gaussifier prediction (Appendix B.1). Each of the three output maps has its own target.

Density. The density target y is a sparse map with a unit mass at each reference Gaussian center and zero elsewhere. We splat each center bilinearly onto its four nearest pixels to keep sub-pixel positions. The sum of y therefore equals the reference count. We train λ with a Poisson loss,

$$\mathcal{L}_{\text{density}} = \frac{1}{P} \sum_{\boldsymbol{x} \in \Omega} \left[\lambda(\boldsymbol{x}) - y(\boldsymbol{x}) \log \lambda(\boldsymbol{x}) \right]. \tag{4}$$

Eq. 4 supervises only the local rate of Gaussian centers and count but does not constrain their spacing, which EMV then sets (Section 3.2).

Log-covariance and color. For each reference Gaussian, we build its covariance from its fitted scale and rotation and take the matrix logarithm. We splat the three entries onto the same four pixels as its center and average them at each pixel with the bilinear weights. This gives the dense target $\bar{S}$. The color target $\bar{C}$ is built the same way from the fitted color coefficients. We do not use pixel colors, because additive rendering makes each pixel a sum of overlapping Gaussians. Both maps use a smooth L_1 loss ℓ averaged with the weights y, so only pixels near reference centers contribute,

$$\mathcal{L}_S = \frac{\sum_{\boldsymbol{x}} y(\boldsymbol{x})\, \ell\big(S(\boldsymbol{x}) - \bar{S}(\boldsymbol{x})\big)}{\sum_{\boldsymbol{x}} y(\boldsymbol{x})}, \tag{5}$$

and $\mathcal{L}_C$ is defined in the same way. The pretraining loss is $\mathcal{L}_{\text{density}} + 0.05\,\mathcal{L}_S + 0.1\,\mathcal{L}_C$.

Reconstruction. Next, we train the refinement network using only images. Our objective supervises the initial reconstruction and the reconstruction after each update,

$$\mathcal{L}_{\text{rec}} = \sum_{t=0}^{T} \alpha_t \,\text{MSE}(\widehat{I}^t, I), \qquad \alpha_t = \frac{t+1}{\sum_{s=0}^{T}(s+1)}. \tag{6}$$

The increasing weights emphasize the later reconstructions. This stage uses about 3.55M training images in Open Images with a fixed 99:1 training and validation split. We also fine-tune the output heads of the pretrained U-Net and keep its other layers frozen (Appendix B.4).

4 Experiments

4.1 Experimental Setup

Datasets. We evaluate on Kodak-24 (Kodak, 1999), which contains 24 images, and on the 100 high-resolution images in the DIV2K validation set (Agustsson & Timofte, 2017). Both datasets use the same preprocessing. We resize the shorter image side to 512 pixels with bilinear interpolation and take a 512×512 center crop.

Baselines and evaluation metrics. We consider two groups of baselines. For feedforward reconstruction, we compare with AIR (Zeng et al., 2026), Instant-GI (Zeng et al., 2025), and Fast-2DGS (Wang et al., 2026). We use their released implementations and checkpoints. We also evaluate Instant-GI* and Fast-2DGS*, obtained by continuing training from their official checkpoints on the same Open Images training images as Gaussifier. For optimization-based methods, we compare with GaussianImage (Zhang et al., 2024b) at a matched Gaussian count (Appendix D.2). We report PSNR and multi-scale SSIM (MS-SSIM) (Wang et al., 2003) averaged over all images, with LPIPS (Zhang et al., 2018) in Appendix D.6. We measure speed and peak device memory on an NVIDIA RTX 5090. Gaussifier runs in its TensorRT/AOTInductor deployment, and the baselines run in FP32. Appendix C also reports compiled baselines.

4.2 Feedforward Reconstruction

Table 1 reports forward-only reconstruction quality and efficiency. Gaussifier achieves the highest PSNR and MS-SSIM on both datasets. With its default adaptive Gaussian count, Gaussifier reaches

Table 1: Feedforward reconstruction on Kodak-24 and DIV2K. Best and second-best results are shown in **bold** and underlined. Mem. is the peak device memory in MB, and #G is the mean Gaussian count in thousands. Higher is better except for Mem., and #G is not ranked. * denotes methods fine-tuned on our training set. Gaussifier uses an image-dependent Gaussian count, and Gaussifier-20k uses a fixed low budget of 20k Gaussians.

	Kodak-24				DIV2K			
Method	PSNR	MS-SSIM	Mem.	#G (k)	PSNR	MS-SSIM	Mem.	#G (k)
Gaussifier	**40.73**	**0.9963**	1,126	42.6	**38.61**	**0.9973**	1,130	72.2
Gaussifier-20k	35.97	0.9860	1,114	20.0	30.15	0.9735	1,114	20.0
AIR	31.86	0.9853	2,852	35.4	27.57	0.9810	2,852	36.3
Instant-GI	30.40	0.9674	3,364	25.1	27.41	0.9714	3,370	28.8
Instant-GI*	32.47	0.9802	3,364	28.8	27.66	0.9799	3,370	29.0
Fast-2DGS	25.95	0.9203	**1,022**	50.0	23.73	0.9233	**1,022**	70.0
Fast-2DGS*	29.03	0.9734	**1,022**	50.0	26.98	0.9607	**1,022**	70.0

40.73/38.61 dB on Kodak-24/DIV2K, exceeding the strongest baseline, Instant-GI*, by 8.26/10.95 dB. Against every released baseline, it has higher PSNR on every image, and paired 95% bootstrap intervals exclude zero (Appendix D.5). This configuration runs in real time at 73/68 FPS and uses 1.1 GB of device memory. Figure 2 presents the qualitative comparison.

Gaussian budget. The Gaussian count of Gaussifier adapts to the image content, from 22k to 85k Gaussians on Kodak-24 and from 19k to 170k on DIV2K. With a set budget on Kodak-24, PSNR rises from 35.97 dB at 20k to 41.56 dB at 50k. At 50k, this matches GaussianImage's 41.37 dB after 20 s of fitting at the same Gaussian count (Appendix D.2). With 20k Gaussians, fewer than any baseline in Table 1 uses, Gaussifier still exceeds the strongest baseline by 3.50/2.49 dB on Kodak-24/DIV2K.

Controlled comparisons. Gaussifier supports a configurable Gaussian count N and number of refinement updates. We compare it with each baseline at the same Gaussian count. At the per-image count of each baseline, Gaussifier improves mean PSNR by 3.0 to 15.6 dB, with higher PSNR on every image (Tables 2 and 9). Additionally, at the same Gaussian count, one or two refinement updates are enough to reach or exceed the mean PSNR of AIR and Instant-GI. Table 2 also compares Gaussifier with baselines that perform per-image fitting at the same counts. Gaussifier reaches its mean PSNR with six refinement updates in at most 14 ms. To reach the same PSNR, AIR and Instant-GI need 0.36 to 1.21 s, and Fast-2DGS needs 4.26 to 4.56 s. This is 26 to 326 times longer than Gaussifier.

Continued optimization. Figure 3 evaluates the predicted representations as initializations for further per-image fitting. Gaussifier uses our fitting pipeline (Appendix D.3), while Instant-GI and Fast-2DGS use their own native refinement procedures. For AIR, we implement a refinement extension with its Predict-Optimize-Distill parameterization and optimization rules. At 0.2 s, Gaussifier reaches 42.94/36.26 dB on Kodak-24/DIV2K. At 20 s, its quality rises to 46.79/40.62 dB. Gaussifier is therefore also a good starting point for further fitting. It also outperforms all three baselines at every measured time up to 20 s.

4.3 Ablation Studies

Refinement depth. Figure 4 shows that average reconstruction quality improves after every update. PSNR rises from 25.40 to 40.73 dB on Kodak-24 and from 22.25 to 38.61 dB on DIV2K. These are total gains of 15.33/16.36 dB over $T = 0$. The gains shrink at later steps, but the last update still adds 0.20/0.26 dB. All of these improvements come from repeatedly updating the same set of Gaussians with a fixed count. Beyond the six trained updates, PSNR still rises by 0.15/0.31 dB, peaking at eight and ten updates, and then slowly declines. Adam started from the same initial $T = 0$ Gaussians needs a median of 110/77 steps to reach the PSNR of six updates (Appendix D.1).

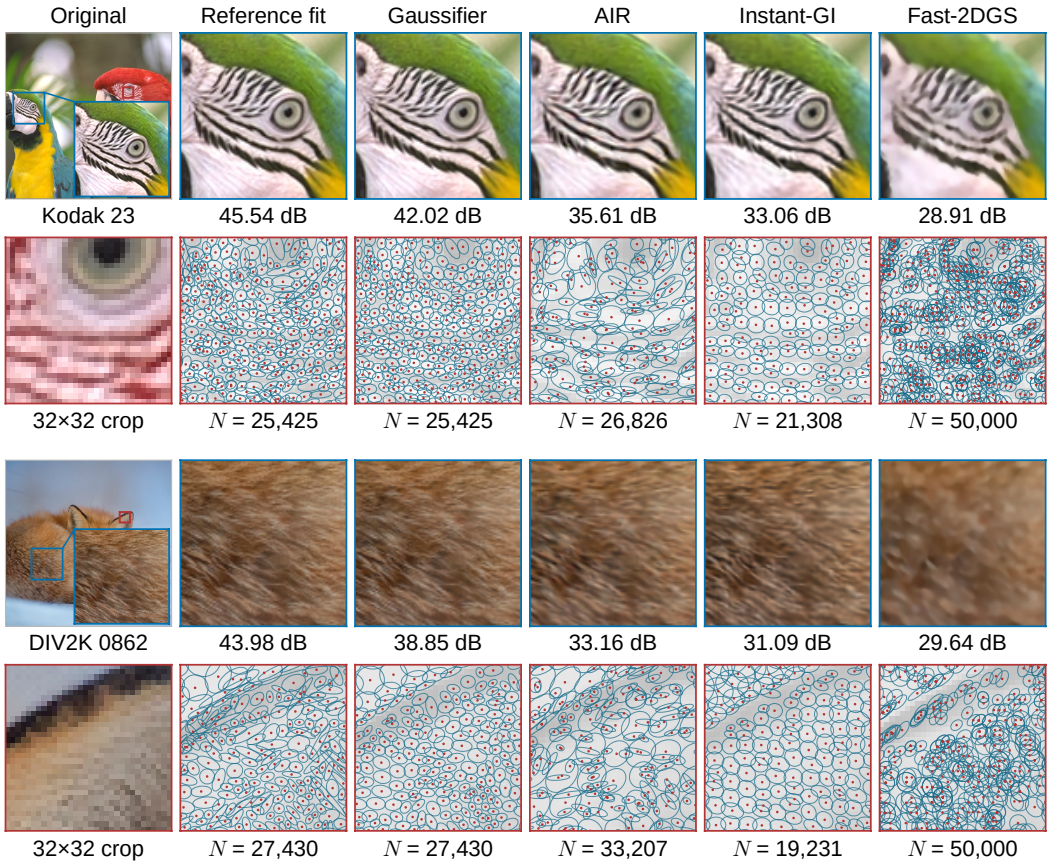

Figure 2: **Feedforward reconstruction quality and Gaussian geometry.** For each example, the first row shows the original image and magnified 96×96 reconstruction crops marked in blue. The crops show a per-image fit for 50k iterations as a reference and the reconstructions of Gaussifier and three feedforward baselines. The second row shows a separate 32×32 crop marked in red, with Gaussian centers (red) and 1σ covariance ellipses (blue). PSNR and the Gaussian count N refer to the full image. Compared with the three baselines, Gaussifier produces more reasonable Gaussian geometry.

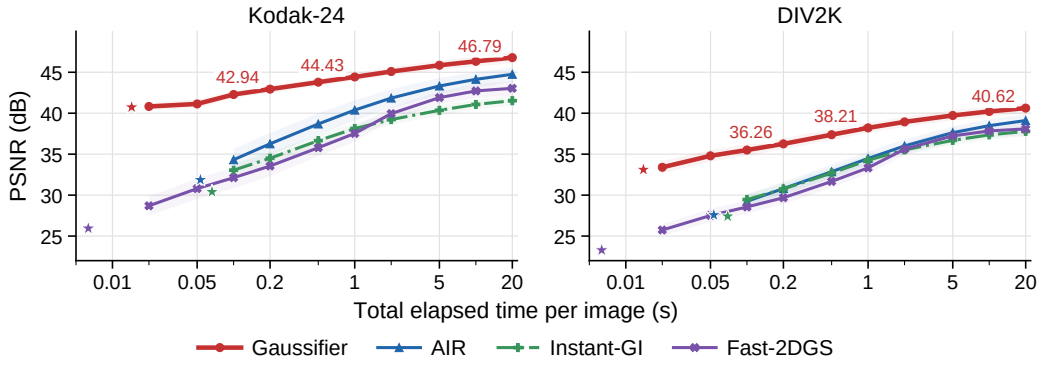

Figure 3: **Quality over time with continued per-image fitting.** On Kodak-24, each method fits at its default Gaussian count (Table 1). On DIV2K, Gaussifier fits at the lowest per-image count among the three baselines. Fast-2DGS uses 50k on both datasets. Stars mark feedforward outputs, and bands show 95% image-bootstrap confidence intervals.

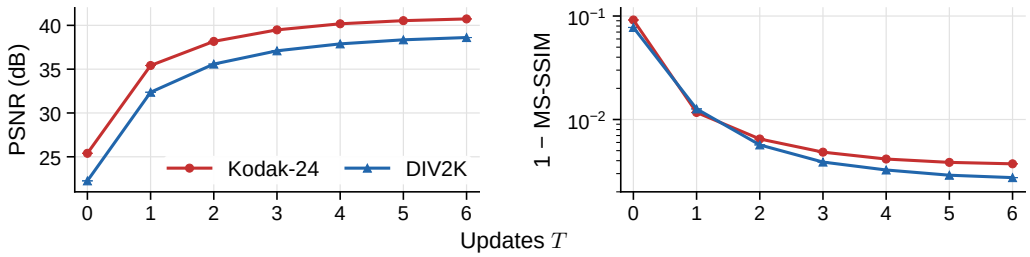

Figure 4: **Quality after each update.** Feedforward reconstruction quality improves as T increases and gradually plateaus beyond $T = 4$. $1 - \text{MS-SSIM}$ is shown on a logarithmic scale.

Table 2: Same-count comparison with the released baselines. Gaussifier uses the per-image count of each baseline, no fitting, and at most 14 ms. Base and Ours are PSNR in dB. Fit is the time a baseline needs with its own fitting to reach our PSNR, with the ratio to 14 ms. Fast-2DGS uses 50k Gaussians. Tables 9 and 10 add AIR ps6/ps7.

	Kodak-24				DIV2K			
Baseline	#G (k)	Base	Ours	Fit	#G (k)	Base	Ours	Fit
AIR	35.4	31.86	**39.86**	0.81 s (58×)	36.3	27.57	**34.90**	1.21 s (86×)
Instant-GI	25.1	30.40	**37.83**	0.87 s (62×)	28.8	27.41	**33.11**	0.60 s (43×)
Fast-2DGS	50.0	25.95	**41.56**	4.26 s (304×)	50.0	23.29	**37.07**	4.56 s (326×)

Density maps. We compare the predicted density with uniform and gradient-guided alternatives. The gradient-guided density follows Image-GS (Zhang et al., 2025), which mixes normalized grayscale Sobel magnitudes with a uniform distribution, using weights 0.7 and 0.3. We keep EMV, the per-image Gaussian count, and the predicted attribute maps unchanged. Table 3 shows that replacing the density with a uniform one loses 5.87/6.13 dB on Kodak-24/DIV2K, and with the gradient-based density loses 1.28/1.80 dB. The trained pipeline therefore relies on the learned density.

Color initialization. We also replace the predicted colors with the input-image RGB, sampled bilinearly at the Gaussian centers, and keep the predicted covariances and density unchanged. This yields 31.81/28.82 dB on Kodak-24/DIV2K after all updates. The trained refinement therefore relies on the predicted color coefficients (Section 3.4).

Refinement network inputs. We examine the input context of $\boldsymbol{U}_\phi$ by removing either the density and count maps or the initial log-covariance and RGB maps, and retraining the refinement network. Removing the density and count maps reduces PSNR by 0.34/0.30 dB on Kodak-24/DIV2K (Table 3). These maps help the network relate predicted representation demand to the current spatial distribution of Gaussian centers. Removing the initial attribute maps reduces PSNR by 0.19/0.10 dB. Since the effect of a parameter update depends on a Gaussian's covariance and color coefficients, the initial attribute maps help the network choose appropriate update magnitudes for a given reconstruction residual. Both input groups provide useful context for refinement, with modest gains in PSNR.

Density samplers. Table 4 compares seven sampling strategies at 512×512 and 1024×1024. EMV achieves the highest PSNR in three of the four settings, trailing Hilbert-Lloyd by only 0.03 dB on DIV2K at 512×512. Relative to Error-Diffusion, EMV improves PSNR by 2.59/2.20 dB on Kodak-24/DIV2K at 512×512, with larger gains of 6.56/3.23 dB at 1024×1024. Poisson and binomial sampling perform substantially worse across all settings. These results support EMV as an effective placement within our trained pipeline.

Table 3: Component ablations. The density and color rows replace one predicted map at test time. The U_ϕ rows retrain U_ϕ without the listed inputs.

	Kodak-24		DIV2K	
Variant	PSNR ↑	MS-SSIM ↑	PSNR ↑	MS-SSIM ↑
Gaussifier (Full)	**40.73**	**0.9963**	**38.61**	**0.9973**
w/o predicted density (uniform)	34.86	0.9875	32.48	0.9853
w/o predicted density (gradient)	39.45	0.9956	36.81	0.9961
w/o predicted RGB initialization	31.81	0.9830	28.82	0.9803
w/o density and count inputs in U_ϕ	40.39	0.9953	38.31	0.9969
w/o initial attribute inputs in U_ϕ	40.54	0.9960	38.51	0.9973

Table 4: Density sampler ablation at two image resolutions. We report RGB PSNR (dB) after $T = 6$ updates, while keeping the trained weights, density, and target number of Gaussians fixed. The weights were fine-tuned with EMV.

	Kodak-24		DIV2K	
Sampler	512×512	1024×1024	512×512	1024×1024
Equal-Mass Voronoi (**ours**)	**40.73**	**46.79**	38.61	**41.25**
Hilbert-Lloyd	40.25	46.71	**38.64**	40.59
Hilbert Equal-Mass	39.81	42.85	37.15	35.59
Hilbert Inverse-CDF	40.37	45.53	38.40	40.67
Error-Diffusion	38.14	40.23	36.41	38.02
Poisson	35.26	35.67	33.66	34.69
Binomial	35.36	35.74	33.83	34.86

5 Limitations and discussion

We evaluate Gaussifier mainly at one image resolution and report speed on one GPU. The cost grows with the image resolution because the refinement network and the renderer run at full resolution in every update. Gaussifier runs at 20 FPS at 1280×720 (Appendix C). Compiled, AIR and Fast-2DGS run as fast as or faster than Gaussifier (Appendix C). However, at the same Gaussian count their outputs are 7 to 16 dB lower (Table 2). Without the predicted density, the same refinement loses about 6 dB (Table 3). The network is trained with six updates, and quality slowly declines after eight to ten steps (Appendix D.1). Gaussifier does not replace long per-image fitting. On Kodak-24, 20 s of fitting from its output adds another 6 dB (Figure 3). Training with more updates could close part of this gap. Future work could add video-specific improvements and fine-tuning that exploit similar adjacent frames. For example, Gaussians could carry over between frames. Other directions are training at higher resolutions and feeding Gaussifier outputs to downstream vision models.

6 Conclusion

Gaussifier shows that high-quality 2D Gaussian decomposition does not require per-image optimization. At the same Gaussian count, it reaches in milliseconds the quality that per-image fitting reaches in seconds. Where to place the Gaussians and how to correct them can both be learned from many images. The Gaussian count and the number of updates then trade speed for quality within one trained model rather than through an optimization run. 2D Gaussians can therefore serve as a representation that real-time systems produce on demand. The same recipe of learning placement from per-image fits and learning corrections across images could also apply to 3D Gaussians.

AI USE STATEMENT

We used generative AI tools to assist with drafting and revising the manuscript, analyzing related literature and their implementation details, and preparing figures and figure-processing scripts. The authors reviewed and revised the AI-assisted text and take responsibility for the final manuscript and associated artifacts.

REPRODUCIBILITY STATEMENT

Section 3 describes the model, allocation algorithm, and training objectives, with architecture, training, and evaluation details in Appendices A, B, and C. Section 4.1 specifies the datasets, preprocessing, baselines, and evaluation protocol. We provide the Gaussifier source code and pretrained weights through an anonymous GitHub repository at `https://github.com/Gaussifier/gaussifier`. A browser-based WebGPU demo runs at `https://gaussifier.github.io/demo/`.

REFERENCES

Eirikur Agustsson and Radu Timofte. Ntire 2017 challenge on single image super-resolution: Dataset and study. In *2017 IEEE conference on computer vision and pattern recognition workshops (CVPRW)*, pp. 1122–1131. IEEE, 2017.

Vincent Arsigny, Pierre Fillard, Xavier Pennec, and Nicholas Ayache. Log-euclidean metrics for fast and simple calculus on diffusion tensors. *Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine*, 56(2):411–421, 2006.

Michael Balzer, Thomas Schlömer, and Oliver Deussen. Capacity-constrained point distributions: A variant of lloyd's method. *ACM Transactions on Graphics (TOG)*, 28(3):1–8, 2009.

David Charatan, Sizhe Lester Li, Andrea Tagliasacchi, and Vincent Sitzmann. pixelsplat: 3d gaussian splats from image pairs for scalable generalizable 3d reconstruction. In *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 19457–19467. IEEE, 2024.

Du Chen, Liyi Chen, Zhengqiang Zhang, and Lei Zhang. Generalized and efficient 2d gaussian splatting for arbitrary-scale super-resolution. In *2025 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 26435–26445. IEEE, 2025.

Tianyu Chen, Wei Xiang, Kang Han, Yu Lu, Di Wu, Gaowen Liu, and Ramana Rao Kompella. Gifsplat: Generative prior-guided iterative feed-forward 3d gaussian splatting from sparse views. *arXiv preprint arXiv:2602.22571*, 2026a.

Yuedong Chen, Haofei Xu, Chuanxia Zheng, Bohan Zhuang, Marc Pollefeys, Andreas Geiger, Tat-Jen Cham, and Jianfei Cai. Mvsplat: Efficient 3d gaussian splatting from sparse multi-view images. In *European conference on computer vision*, pp. 370–386. Springer, 2024.

Yuhan Chen, Ying Fang, Guofa Li, Wenxuan Yu, Yicui Shi, Jingrui Zhang, Kefei Qian, Wenbo Chu, and Keqiang Li. Ll-gaussianmap: Zero-shot low-light image enhancement via 2d gaussian splatting guided gain maps. *arXiv preprint arXiv:2601.15766*, 2026b.

Yuhan Chen, Wenxuan Yu, Guofa Li, Kunyang Huang, Ying Fang, Yicui Shi, Wenbo Chu, and Keqiang Li. Dehaze-gaussianimage: Zero-shot dehazing via efficient 2d gaussian splatting representation. *arXiv preprint arXiv:2606.16163*, 2026c.

Yuhan Chen, Wenxuan Yu, Guofa Li, Yijun Xu, Ying Fang, Yicui Shi, Long Cao, Wenbo Chu, and Keqiang Li. Ll-gaussianimage: Efficient image representation for zero-shot low-light enhancement with 2d gaussian splatting. *arXiv preprint arXiv:2601.15772*, 2026d.

Fernando De Goes, Katherine Breeden, Victor Ostromoukhov, and Mathieu Desbrun. Blue noise through optimal transport. *ACM Transactions on Graphics (TOG)*, 31(6):1–11, 2012.

Oliver Deussen, Marc Spicker, and Qian Zheng. Weighted linde-buzo-gray stippling. *ACM Transactions on Graphics (TOG)*, 36(6):1–12, 2017.

Jiajun Dong, Chengkun Wang, Wenzhao Zheng, Lei Chen, Jiwen Lu, and Yansong Tang. Gaussiantoken: An effective image tokenizer with 2d gaussian splatting. *arXiv preprint arXiv:2501.15619*, 2025.

Qiang Du, Vance Faber, and Max Gunzburger. Centroidal voronoi tessellations: Applications and algorithms. *SIAM review*, 41(4):637–676, 1999.

Emilien Dupont, Adam Goliński, Milad Alizadeh, Yee Whye Teh, and Arnaud Doucet. Coin: Compression with implicit neural representations. *arXiv preprint arXiv:2103.03123*, 2021.

Jefferson Hernandez, Ruozhen He, Guha Balakrishnan, Alexander C Berg, and Vicente Ordonez. Gvit: Representing images as gaussians for visual recognition. *arXiv preprint arXiv:2506.23532*, 2025.

David Hilbert. Über die stetige abbildung einer linie auf ein flächenstück. In *Dritter Band: Analysis· Grundlagen der Mathematik· Physik Verschiedenes: Nebst Einer Lebensgeschichte*, pp. 1–2. Springer, 1935.

Jintong Hu, Bin Xia, Bin Chen, Wenming Yang, and Lei Zhang. Gaussiansr: High fidelity 2d gaussian splatting for arbitrary-scale image super-resolution. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 3554–3562, 2025.

Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, George Drettakis, et al. 3d gaussian splatting for real-time radiance field rendering. *ACM Trans. Graph.*, 42(4):139–1, 2023.

Kodak. Kodak lossless true color image suite, 1999. URL `https://r0k.us/graphics/kodak/`.

Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Alexander Kolesnikov, et al. The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale. *International journal of computer vision*, 128(7):1956–1981, 2020.

Victor Lempitsky and Andrew Zisserman. Learning to count objects in images. *Advances in neural information processing systems*, 23, 2010.

Tiantian Li, Xinjie Zhang, Xingtong Ge, Tongda Xu, Dailan He, Jun Zhang, and Yan Wang. Gaussianimage++: Boosted image representation and compression with 2d gaussian splatting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 6442–6449, 2026.

Huanxiong Liang, Yunuo Chen, Yicheng Pan, Sixian Wang, Jincheng Dai, Guo Lu, and Wenjun Zhang. Structure-guided allocation of 2d gaussians for image representation and compression. *arXiv preprint arXiv:2512.24018*, 2025.

Yehe Liu, Alexander Krull, Hector Basevi, Aleš Leonardis, and Michael Jenkins. bit2bit: 1-bit quanta video reconstruction via self-supervised photon prediction. *Advances in Neural Information Processing Systems*, 37:88443–88485, 2024.

Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2): 129–137, 1982.

Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM transactions on graphics (TOG)*, 41(4): 1–15, 2022.

Yasmine Omri, Connor Ding, Tsachy Weissman, and Thierry Tambe. Gaussianvision: Vision-language alignment from compressed image representations using 2d gaussian splatting. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 14926–14935, 2026.

Guodong Rong and Tiow-Seng Tan. Jump flooding in gpu with applications to voronoi diagram and distance transform. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pp. 109–116, 2006.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pp. 234–241. Springer, 2015.

Vincent Sitzmann, Julien Martel, Alexander Bergman, David Lindell, and Gordon Wetzstein. Implicit neural representations with periodic activation functions. *Advances in neural information processing systems*, 33:7462–7473, 2020.

Stanislaw Szymanowicz, Chrisitian Rupprecht, and Andrea Vedaldi. Splatter image: Ultra-fast single-view 3d reconstruction. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 10208–10217, 2024.

Joanna Waczyńska, Tomasz Szczepanik, Piotr Borycki, Sławomir Tadeja, Thomas Bohné, and Przemysław Spurek. Mirage: Editable 2d images using gaussian splatting. *arXiv preprint arXiv:2410.01521*, 2024.

Hao Wang, Ashish Bastola, Chaoyi Zhou, Wenhui Zhu, Xiwen Chen, Xuanzhao Dong, Siyu Huang, and Abolfazl Razi. Fast 2dgs: Efficient image representation with deep gaussian prior. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pp. 1184–1193, 2026.

Longan Wang, Yuang Shi, and Wei Tsang Ooi. Gsvc: Efficient video representation and compression through 2d gaussian splatting. In *Proceedings of the 35th Workshop on Network and Operating System Support for Digital Audio and Video*, pp. 15–21, 2025.

Zhou Wang, Eero P Simoncelli, and Alan C Bovik. Multiscale structural similarity for image quality assessment. In *The thrity-seventh asilomar conference on signals, systems & computers, 2003*, volume 2, pp. 1398–1402. Ieee, 2003.

Haofei Xu, Daniel Barath, Andreas Geiger, and Marc Pollefeys. Resplat: Learning recurrent gaussian splatting. In *European Conference on Computer Vision*, pp. 364–382. Springer, 2026.

Zhaojie Zeng, Yuesong Wang, Tao Guan, Chao Yang, and Lili Ju. Instant gaussianimage: A generalizable and self-adaptive image representation via 2d gaussian splatting. In *2025 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 27896–27905. IEEE, 2025.

Zhaojie Zeng, Yuesong Wang, Yawei Luo, and Tao Guan. Air: Amortized image reconstruction framework for self-supervised feed-forward 2d gaussian splatting. *arXiv preprint arXiv:2605.20820*, 2026.

Kai Zhang, Sai Bi, Hao Tan, Yuanbo Xiangli, Nanxuan Zhao, Kalyan Sunkavalli, and Zexiang Xu. Gs-lrm: Large reconstruction model for 3d gaussian splatting. In *European Conference on Computer Vision*, pp. 1–19. Springer, 2024a.

Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *2018 IEEE/CVF conference on computer vision and pattern recognition*, pp. 586–595. IEEE, 2018.

Xinjie Zhang, Xingtong Ge, Tongda Xu, Dailan He, Yan Wang, Hongwei Qin, Guo Lu, Jing Geng, and Jun Zhang. Gaussianimage: 1000 fps image representation and compression by 2d gaussian splatting. In *European Conference on Computer Vision*, pp. 327–345. Springer, 2024b.

Yunxiang Zhang, Bingxuan Li, Alexandr Kuznetsov, Akshay Jindal, Stavros Diolatzis, Kenneth Chen, Anton Sochenov, Anton Kaplanyan, and Qi Sun. Image-gs: Content-adaptive image representation via 2d gaussians. In *Proceedings of the Special Interest Group on Computer Graphics and Interactive Techniques Conference Conference Papers*, pp. 1–11, 2025.

Zhengqiang Zhang, Rongyuan Wu, Lingchen Sun, and Lei Zhang. Gpstoken: Gaussian parameterized spatially-adaptive tokenization for image representation and generation. *Advances in Neural Information Processing Systems*, 38:11218–11249, 2026.

A METHOD DETAILS

A.1 ARCHITECTURE AND PARAMETERIZATION

The image backbone is a residual U-Net with base width 32 and four resolution levels. Three 1×1 prediction heads follow it. The heads output one log-density channel, three independent entries of a symmetric log-covariance matrix, and three sigmoid-activated RGB channels. The refinement U-Net has two parallel branches with base widths 32 and 24. Each branch has two downsampling stages and receives the same context. Their outputs are summed. Each branch predicts eight update channels, with three for log-covariance, three for RGB, and two for position. The complete model has 2,802,895 parameters, and 746,792 of them belong to the refinement module.

The refinement input has 17 channels. They are the input RGB image, the mean-normalized density, the initial log-covariance map, the initial RGB map, the current RGB reconstruction, the RGB residual, and a single-channel count raster of the centers. Each center adds one count to the pixel that contains it. This raster is recomputed after position updates. The initial attribute maps and the cell-mass weights remain fixed. The evolving Gaussian parameters and their rendering carry the information between steps.

The main text uses pixel coordinates for spatial quantities. The implementation stores centers in $[0, 1]^2$ and covariances in pixel units, and it converts the centers before rasterization. For a raw two-channel position prediction $\boldsymbol{d}_i^t$, the normalized displacement is $2.56\boldsymbol{d}_i^t$ divided by the longer image side. The increment $\Delta\boldsymbol{\mu}_i^t$ in Eq. 3 includes this fixed coordinate scaling. Covariance updates act on the three independent entries of the symmetric matrix before exponentiation. The density equations in the main text omit numerical clipping and small stability constants. Count prediction evaluates the exponential after clipping the logits to $[-20, 8]$. Training caps the count at $N_{\max} = \min(10^6, \lfloor 0.4P \rfloor)$. Evaluation uses the predicted count without this cap.

A.2 APPROXIMATE EQUAL-MASS PLACEMENT

The initial sites come from stratified sampling over randomly shifted tiles. Each tile receives a share of the sites proportional to its density mass. Error diffusion over the tiles rounds these shares to integers. Inside a tile, sites are drawn at distinct pixels in proportion to ρ and jittered within the pixel. Placement is deterministic for a fixed seed. Across seeds 0, 1, and 2, the mean PSNR after six updates has a standard deviation of 0.007 and 0.001 dB on Kodak-24 and DIV2K. Each merge round removes at most a third of the sites. Competing proposals to the same target are resolved by source mass, and a losing source tries its next available neighbor. Jump flooding (Rong & Tan, 2006) gives approximate nearest-center assignments on the pixel grid. Candidates with no assigned mass are removed before matching. Additional merges enforce the requested final count. Lloyd updates alone do not impose equal capacities. The training code checks for degenerate cell configurations. In that case, it can fall back to an equal-mass Hilbert partition followed by Lloyd updates. The final training pass promoted checkpoints only when no fallback occurred, and the released inference runtime has no fallback. On Kodak-24 and DIV2K, 0.002% of the cells are empty and 0.001% of the centers are duplicates. On an RTX 5090, placement takes 1.84 and 2.53 ms on Kodak-24 and DIV2K, and 2.06 and 2.75 ms with the cell-mass weights.

EMV reduces the spread of the cell masses but does not make them exactly equal. The coefficient of variation of $m_i/\bar{m}$ is 0.47 and 0.46 on Kodak-24 and DIV2K when N sites are drawn directly by stratified sampling. It is 0.31 and 0.35 after merging, and 0.27 and 0.32 after the three Lloyd iterations. Cells average 6.2 and 3.6 pixels, so pixel quantization adds part of this spread.

The final cell assignments are used to compute the cell-mass weights w_i. These assignments are fixed in the backward pass. However, the density accumulation within each cell remains differentiable. The weights therefore provide a gradient for the shape of the normalized density. They do not differentiate the integer count or the movement of cell boundaries. They are also invariant to a global rescaling of the density when the count and assignments are fixed. Absolute count supervision and the calibration in Appendix B.3 provide the complementary budget signal. Setting all weights to one at inference lowers the initial PSNR by 3.9 and 2.8 dB on Kodak-24 and DIV2K. After six updates, the loss shrinks to 0.22 and 0.29 dB.

A.3 Fixed Gaussian Budgets

The density head predicts an image-specific count N_{auto}, and the covariance map is learned for that count. A user can instead request a budget N. EMV then places N centers from the same normalized density ρ. Rendering is additive (Eq. 1). Pinning N above N_{auto} without adjusting the covariances therefore brightens the initial rendering roughly in proportion to N/N_{auto}. Far above the predicted count, most pixels saturate. For $N > N_{\text{auto}}$, we therefore scale every initial covariance by N_{auto}/N. We add $\log(N_{\text{auto}}/N)$ to the diagonal of $\boldsymbol{S}_i^0$, which keeps the total area covered by the Gaussians constant. Below N_{auto}, the covariances are left unchanged. Refinement recovers the dimmer initial rendering. Enlarging the covariances there changed the final PSNR by -0.33 to $+0.40$ dB, depending on the count. The rule applies only at inference, adds no parameters, and does not affect the adaptive-count results.

B Training Details

B.1 Reference Fits

We use two sets of Open Images data. Open Images is not an evaluation dataset in this work. The first set has about 107k images from the Open Images test split. They are fitted for 2,800 steps with our fitting pipeline. It optimizes the centers, scales, rotations, and colors of all Gaussians with Adam on the MSE through the additive renderer. Its adaptive density control periodically moves Gaussians that leave the image, cover no pixels, or grow very large to pixels with high error. A ridge regression on patch gradient, Laplacian, and color-variance statistics sets the initial count. Images below 38 dB are refit once with twice the Gaussians, and the better fit is kept. These fits have a median of 64k Gaussians and 42.3 dB PSNR. Their optimization took 326 GPU-hours in total. The second set has about 20k images from the Open Images train split. Their fits start from an earlier Gaussifier prediction and are optimized for 2,400 steps. These fits took about 33 GPU-hours.

B.2 Training Stages

The released model comes from a sequence of runs. Pretraining on the first set uses the density loss and the covariance and color losses with weights 0.1 and 1.0. It adds a small reconstruction loss in its last epochs. The refinement network is then trained on the same images with one, two, and then four updates. These runs first use the MSE and then an L_1 and SSIM loss. Some of them also update the U-Net. Some add losses toward the render and the centers refined by 100 Adam steps from the current Gaussians. The U-Net is then fine-tuned on the second set with the losses of Section 3.4 and weights 0.05 and 0.1. The refinement network continues on the second set and then on images alone, with four and then six updates. Three later stages update the output heads and the refinement network. One of them also updates the backbone at a learning rate of 3×10^{-8}. These stages move each center to the centroid of its cell before rendering and pass gradients to the density through soft centroids. Their count losses raise the predicted count on images with low PSNR and lower it on images with very high PSNR. Another loss moves density toward tiles with large residuals. The next stage adapts the output heads to Hilbert-Lloyd placement (Appendix C.3). The refinement network is then trained with the increasing and the final-state MSE, and the two results are averaged with weights 0.75 and 0.25. Count calibration (Appendix B.3) and the final adaptation (Appendix B.4) follow. These stages add no networks or count search at inference. Table 5 lists the stages and their wall-clock time.

B.3 Count Calibration

We calibrate the density head on 4,096 training images with a small discrete search around the predicted budget of each image. The candidate multipliers are 0.9, 1.0, and 1.1. Among the candidates that reach a 41 dB quality threshold, we select the smallest count. If none reaches the threshold, we select the candidate with the highest PSNR. An increase is accepted only when it improves the measured reconstruction quality over the original budget. These targets are local and empirically evaluated. We do not claim that they are globally optimal Gaussian counts.

Table 5: Training stages of the released model and their wall-clock time on one RTX 5090. The times include the steps after the retained checkpoints.

Stage	Data	Trained parts	Hours
Pretraining	First set	U-Net	4.3
Refinement with one to four updates	First set	Refinement, partly the U-Net	47.9
Fine-tuning and paired refinement	Second set	U-Net, then refinement	27.7
Refinement with four, then six updates	Images only	Refinement	153.4
Centroid, count, and local-density losses	Images only	Heads, refinement, briefly the backbone	20.8
Hilbert-Lloyd adaptation and MSE catch-up	Images only	Heads, then refinement	6.5
Count calibration	4,096 images	Density head	<1
Final adaptation	Images only	Heads and refinement	166.6
Total			428

Let $N^\star$ denote a selected target and $N_\theta = \sum_{\boldsymbol{x}\in\Omega} \lambda_\theta(\boldsymbol{x})$ the predicted count. Let $p_\theta = \rho_\theta / P$ and p_{ref} be the predicted and pre-calibration spatial distributions. The calibration objective is

$$\mathcal{L}_{\mathrm{cal}} = (\log N_\theta - \log N^\star)^2 + \beta D_{\mathrm{KL}}(p_{\mathrm{ref}} \,\|\, p_\theta), \qquad \beta = 4. \tag{7}$$

For examples with an unchanged budget, the regression target keeps the original unrounded density sum. This prevents count caps from causing a systematic downward bias. Only the 33 parameters of the density head are updated in this stage. At inference, the sum of the predicted density gives the count directly.

B.4 Final Adaptation

The final stage uses a batch size of two and the increasing-state MSE objective in Eq. 6. The image backbone is frozen. AdamW updates the refinement module at an initial learning rate of 5×10^{-8}. It updates the density and attribute heads at one tenth of this rate, with zero weight decay. A cosine schedule uses 5,000 warmup steps and a minimum refinement learning rate of 5×10^{-9}. We select the retained checkpoint at step 540,000, after 1,080,000 image presentations. The selection uses 256 images from the validation part of the 99:1 split in Section 3.4. Its criterion emphasizes the lower tail of reconstruction quality. No Kodak-24 or DIV2K image is used for checkpoint selection or count calibration. Its sampler uses hard positions and assignments, and density gradients pass through the cell-mass weights.

C Evaluation Details

C.1 Protocol

All timings use one NVIDIA RTX 5090 with batch size one. Gaussifier runs its C++ deployment with a TensorRT FP16 U-Net, an FP16 refinement network compiled with PyTorch AOTInductor (AOTI), fused update kernels, and seed 12345. Each image runs three warm-up passes and 30 timed passes with synchronized wall-clock timing. We take the median time of each image and average it over the dataset. FPS is 1,000 divided by this average in milliseconds. Memory in Table 1 is the peak device memory of each process, as reported by NVML. It includes the CUDA context and the memory of every library. AIR and Instant-GI use seed 0. For Fast-2DGS, quality is averaged over seeds 0, 1, and 2, and timing uses seed 0. The baselines run their released code in eager PyTorch FP32. Table 6 adds latency percentiles, eager Gaussifier, and the peak of the PyTorch or LibTorch allocator. The allocator peak excludes the CUDA context and memory managed by TensorRT or CuPy. In eager PyTorch FP32, Gaussifier takes 49.6/50.6 ms and is about as fast as AIR. Table 7 reports Gaussifier on DIV2K at 1280×720 and 1920×1080. Its cost grows with the number of pixels and Gaussians. It runs at 20 and 8 FPS at these sizes.

C.2 Baselines

AIR uses its released final full-precision stage with codec execution disabled. AIR releases checkpoints for patch sizes 5, 6, and 7 (ps5, ps6, and ps7). Smaller patches give more Gaussians. Table 1

Table 6: Runtime details at 512×512 on an RTX 5090. Latency is the median of 30 timed passes for each image, summarized over the images by the mean, the 95th percentile, and the maximum. Allocated memory is the peak of the PyTorch or LibTorch allocator. Device memory is the NVML peak of the process, as in Table 1. Memory is given as Kodak-24/DIV2K. The compiled baselines keep their released checkpoints and use torch.compile with CUDA graphs in FP16, or in BF16 for Instant-GI, whose activations overflow FP16. The rasterizers and the CuPy triangulation of Instant-GI stay as released. Compilation changes their PSNR by at most 0.01 dB. CUDA graphs hide part of their allocation, so we omit it.

		Latency (ms), mean / p95 / max			
Method	Runtime	Kodak-24	DIV2K	Allocated MB	Device MB
Gaussifier	TensorRT/AOTI FP16	13.7 / 14.5 / 14.9	14.7 / 16.4 / 23.8	180/183	1,126/1,130
Gaussifier	PyTorch FP32	49.6 / 50.8 / 50.9	50.6 / 52.2 / 57.4	448/449	1,304/1,306
AIR	PyTorch FP32	53.0 / 53.2 / 53.2	53.2 / 53.3 / 53.4	1,912/1,912	2,852/2,852
Instant-GI	PyTorch FP32, CuPy	65.4 / 75.4 / 79.2	68.7 / 80.6 / 84.7	999/999	3,364/3,370
Fast-2DGS	PyTorch FP32	6.3 / 6.3 / 6.4	6.4 / 6.4 / 6.5	310/310	1,022/1,022
AIR	compiled FP16	12.3 / 12.4 / 12.4	12.4 / 12.4 / 12.4	–	2,460/2,460
Instant-GI	compiled BF16, CuPy	31.6 / 36.9 / 38.4	32.4 / 38.6 / 41.5	–	3,010/3,364
Fast-2DGS	compiled FP16	1.7 / 1.7 / 1.7	1.8 / 1.8 / 1.9	–	1,224/1,238

Table 7: Gaussifier on DIV2K at larger frame sizes on an RTX 5090, with the TensorRT and AOTI modules built for each size. Each image is resized to cover the frame and center-cropped. Latency and memory are measured as in Table 6.

Frame size	#G (k)	Latency (ms), mean / p95 / max	FPS	PSNR	Allocated MB	Device MB
512×512	72.2	14.7 / 16.4 / 23.8	68	38.61	183	1,130
1280×720	163.0	51.1 / 59.1 / 75.8	20	40.35	612	2,264
1920×1080	243.1	125.2 / 133.9 / 149.8	8	42.01	1,348	4,132

reports ps5, and Table 9 also reports ps6 and ps7. Instant-GI uses its released checkpoint with kernel size 3 (ks3). This setting max-pools the position map over 3×3-pixel blocks before dithering. Instant-GI runs with its CuPy Delaunay and native dithering and ellipse extensions. Fast-2DGS uses its released heatmap-prior checkpoint and its attribute checkpoint with offset refinement (Plus). It places 50k Gaussians on Kodak-24 and 70k on DIV2K. Instant-GI* and Fast-2DGS* continue training from these official checkpoints on the Open Images training images of Section 3.4.

C.3 Sampler Ablation

Table 4 keeps the trained weights, the density, and the count fixed and changes only the placement. The cell-mass weights are computed in the same way for every sampler. Hilbert Equal-Mass orders the pixels along a Hilbert curve (Hilbert, 1935) and cuts the cumulative mass into N equal segments. It places a center at the mass centroid of each segment. Hilbert-Lloyd adds three Lloyd iterations to these centers. Hilbert Inverse-CDF places the N centers by inverting the cumulative distribution function (CDF) of the density along the same curve. Error-Diffusion draws N centers directly with the stratified sampler of Appendix A.2, without merging or Lloyd iterations. Poisson and Binomial draw a count for each pixel, with eight trials for Binomial, and jitter the points within the pixel. Their total count therefore varies around N.

D Additional Results

D.1 Refinement Checks

These checks follow the protocol of Table 1 in eager FP32 PyTorch with seed 12345 and the production cell-mass weights. With the same weights, PSNR after 6, 8, 10, and 12 updates is 40.74, 40.89, 40.79, and 40.55 dB on Kodak-24. On DIV2K, it is 38.61, 38.87, 38.92, and 38.82 dB.

For the Adam comparison, Adam starts from the initial Gaussians of the same run. It optimizes the same centers, log-covariances, and colors with the same renderer, weights, and projections as Eq. 3. It uses the MSE and constant learning rates. The learning rates are 4×10^{-4} for the normalized centers and 0.02 for the log-covariances and colors, the best of six scales on four Kodak-24 and four DIV2K images. Tuning on test images favors Adam. Six learned updates take 42 ms in this implementation, and one Adam step takes about 0.9 to 1.0 ms. At equal time, Adam is 2.1 and 2.5 dB lower on Kodak-24 and DIV2K. Adam reaches the PSNR of six updates after a median of 110 steps on Kodak-24 and 77 steps on DIV2K. Four DIV2K images do not reach it within 1,000 steps. After 1,000 steps, Adam reaches 44.40 and 43.87 dB.

D.2 GaussianImage at a Matched Count

This comparison supports the Gaussian budget result in Section 4.2. We fit GaussianImage (Zhang et al., 2024b) to all 24 Kodak images with the preprocessing of Section 4.1. It starts from its native initialization, and its schedule is adjusted to a final budget of 50k Gaussians. After 20 s of fitting on an NVIDIA L40S GPU, it reaches 41.37 dB. Gaussifier reaches 41.56 dB with the same 50k Gaussians and no fitting.

D.3 Continued Optimization

For Figure 3, our fitting pipeline (Appendix B.1) starts from the final Gaussifier output. On DIV2K, it starts from Gaussifier at the lowest per-image count among the three baselines, 28.8k on average. This count is below the predicted count on 98 of the 100 images, where the covariances stay unchanged (Appendix A.3). Its adaptive density control is disabled. Its learning rates warm up for 25 steps and then decay exponentially over a 50k-step horizon. The reference fits in Figure 2 use the same setting for 50k steps. For AIR, we apply its Predict-Optimize-Distill (POD) optimizer to all accumulated Gaussians at once. This extension keeps the Adam optimizer, the learning rate of 0.001, and the $0.7L_1 + 0.3(1 - \mathrm{SSIM})$ loss of AIR. Instant-GI and Fast-2DGS use their released fitting code with the Adan and AdamW optimizers. Table 8 lists the values behind Figure 3.

Table 8: PSNR (dB) after a given total elapsed time per image on an RTX 5090. The time includes initialization, the transfer to the optimizer, and per-image fitting, and the runs are those of Figure 3, plus AIR ps6 and ps7 for Table 10. Gaussifier is refined with our fitting pipeline, with adaptive density control disabled. Instant-GI and Fast-2DGS use their native refinement, and AIR uses our POD-based refinement extension. A dash marks budgets at which a pipeline has not yet produced a result for every image. On DIV2K, Gaussifier fits at the lowest per-image count among AIR, Instant-GI, and Fast-2DGS (Appendix D.3). Without any fitting, Gaussifier reaches 40.73 dB on Kodak in 13.7 ms (Table 1), and the best competing pipeline stays below this until after 1 s.

Pipeline	0.02 s	0.05 s	0.1 s	0.2 s	0.5 s	1 s	2 s	5 s	10 s	20 s
Kodak-24										
Gaussifier	**40.82**	**41.13**	**42.28**	**42.94**	**43.79**	**44.43**	**45.09**	**45.85**	**46.33**	**46.79**
AIR ps5	–	–	34.32	36.26	38.69	40.38	41.85	43.32	44.14	44.75
AIR ps6	–	–	33.30	34.80	36.64	37.96	39.11	40.24	40.80	41.17
AIR ps7	–	–	32.55	33.73	35.17	36.22	37.14	38.00	38.41	38.66
Instant-GI	–	–	33.04	34.50	36.65	38.12	39.22	40.35	41.07	41.53
Fast-2DGS	28.70	30.80	32.13	33.55	35.79	37.53	39.95	41.91	42.72	43.04
DIV2K										
Gaussifier	**33.40**	**34.80**	**35.50**	**36.26**	**37.37**	**38.21**	**38.94**	**39.72**	**40.21**	**40.62**
AIR ps5	–	–	29.25	30.80	32.86	34.47	36.01	37.62	38.48	39.10
AIR ps6	–	–	28.57	29.75	31.25	32.40	33.47	34.53	35.03	35.34
AIR ps7	–	–	28.05	28.99	30.16	31.05	31.87	32.62	32.93	33.10
Instant-GI	–	–	29.46	30.75	32.69	34.26	35.52	36.69	37.35	37.79
Fast-2DGS	25.74	27.51	28.55	29.68	31.67	33.33	35.61	37.24	37.83	38.08

D.4 MATCHED COMPARISONS

Tables 9 and 10 give the per-baseline results of the controlled comparisons in Section 4.2.

Table 9: Matched feedforward comparisons with the released baselines at the same Gaussian count. We re-ran all methods on the same RTX 5090 under the protocol of Table 1. Fast-2DGS uses 50k Gaussians on Kodak and 70k on DIV2K. Gaussifier uses the per-image count of each baseline. In the **Same count** column, it runs all $T{=}6$ updates, and it has the higher PSNR on every image. In the **Same count and PSNR** columns, it uses the fewest updates whose mean PSNR reaches that of the baseline. This is $T{=}1$, and $T{=}2$ for AIR ps7 on DIV2K. Since T is discrete, the PSNR reached can exceed that of the baseline. For Fast-2DGS, one update already exceeds its PSNR by 8.3 to 9.5 dB, so we do not compare times. #G is the number of Gaussians. Times are batch-one synchronized wall times in ms, given as the mean of per-image medians over 30 repetitions. Gaussifier runs in its TensorRT/AOTI deployment, and the baselines run in their compiled implementations (Table 6). Gaussifier's PSNR comes from its eager PyTorch implementation, which matches the deployment within 0.02 dB. At the fixed counts of Fast-2DGS, it comes from the deployment with the covariance rule of Appendix A.3.

	Baseline			Same count	Same count and PSNR	
Baseline	#G	ms	PSNR	Our PSNR	Our ms	Our PSNR
Kodak-24						
AIR ps5	35,386	12.3	31.86	39.86 (+8.0)	4.65 (2.7×)	34.75
AIR ps6	25,468	12.4	31.16	37.84 (+6.7)	4.41 (2.8×)	32.94
AIR ps7	19,260	12.4	30.60	35.70 (+5.1)	4.27 (2.9×)	30.65
Instant-GI	25,126	31.6	30.40	37.83 (+7.4)	4.45 (7.1×)	32.83
Fast-2DGS	50,000	1.7	25.95	41.56 (+15.6)	–	–
DIV2K						
AIR ps5	36,294	12.4	27.57	34.90 (+7.3)	4.62 (2.7×)	29.59
AIR ps6	25,706	12.4	27.10	32.14 (+5.0)	4.38 (2.8×)	27.25
AIR ps7	19,260	12.4	26.71	29.74 (+3.0)	6.08 (2.0×)	28.16
Instant-GI	28,825	32.4	27.41	33.11 (+5.7)	4.46 (7.3×)	28.06
Fast-2DGS	70,000	1.8	23.73	38.78 (+15.1)	–	–

Table 10: Baselines with their own per-image fitting versus Gaussifier without fitting, at the same Gaussian count. Each baseline starts from its released feedforward prediction and applies its native refinement, which keeps its count fixed. For AIR, we use our POD-based extension. The "Time" column gives the elapsed time at which the mean PSNR of the baseline reaches the feedforward PSNR of Gaussifier at that count. We interpolate it logarithmically between the budgets of Table 8. Gaussifier takes at most 14 ms, and the speedups use this value. The "Reached" column counts the images whose own curve reaches the PSNR of Gaussifier within 20 s. The fitting runs use 50k Gaussians for Fast-2DGS on both datasets, on an RTX 5090. Our PSNR is measured as in Table 9.

Dataset	Baseline	# Gaussians	Our PSNR	Time	Speedup	Reached
Kodak-24	AIR ps5	35,386	39.86	0.81 s	58×	24/24
	AIR ps6	25,468	37.84	0.94 s	67×	24/24
	AIR ps7	19,260	35.70	0.71 s	51×	24/24
	Instant-GI	25,126	37.83	0.87 s	62×	24/24
	Fast-2DGS	50,000	41.56	4.26 s	304×	22/24
DIV2K	AIR ps5	36,294	34.90	1.21 s	86×	100/100
	AIR ps6	25,706	32.14	0.85 s	61×	99/100
	AIR ps7	19,260	29.74	0.36 s	26×	98/100
	Instant-GI	28,825	33.11	0.60 s	43×	100/100
	Fast-2DGS	50,000	37.07	4.56 s	326×	71/100

D.5 Paired Statistics

Table 11 pairs the per-image PSNR of the methods in Tables 1 and 3. The PSNR of each method is first averaged over its seeds for each image. The intervals are 95% percentile bootstrap intervals over images with 10,000 resamples. The fine-tuned baselines and the retrained rows of Table 3 are not included.

Table 11: Paired PSNR differences in dB, with 95% bootstrap intervals and the number of images on which the first method has the higher PSNR.

	Kodak-24			DIV2K		
Comparison	Diff.	95% interval	Wins	Diff.	95% interval	Wins
Gaussifier vs. AIR	+8.87	[7.71, 10.07]	24/24	+11.04	[10.44, 11.63]	100/100
Gaussifier vs. Instant-GI	+10.34	[9.64, 11.06]	24/24	+11.20	[10.73, 11.65]	100/100
Gaussifier vs. Fast-2DGS	+14.78	[13.73, 15.92]	24/24	+14.88	[14.33, 15.44]	100/100
Gaussifier-20k vs. AIR	+4.10	[3.60, 4.60]	24/24	+2.58	[2.22, 2.94]	95/100
Gaussifier-20k vs. Instant-GI	+5.57	[4.64, 6.47]	24/24	+2.74	[2.26, 3.22]	90/100
Gaussifier-20k vs. Fast-2DGS	+10.02	[9.30, 10.73]	24/24	+6.42	[6.01, 6.85]	100/100
Full vs. uniform density	+5.87	[4.52, 7.28]	24/24	+6.13	[5.27, 7.01]	100/100
Full vs. gradient density	+1.28	[1.03, 1.55]	24/24	+1.80	[1.54, 2.09]	98/100
Full vs. RGB initialization	+8.93	[7.68, 10.25]	24/24	+9.79	[9.00, 10.58]	98/100

D.6 Perceptual Quality

Table 12 reports LPIPS (Zhang et al., 2018) with the AlexNet backbone for the methods of Table 1. Gaussifier has the lowest LPIPS on both datasets. With 20k Gaussians, it is on par with AIR on Kodak-24 and better on DIV2K.

Table 12: LPIPS with the AlexNet backbone for the feedforward methods of Table 1. Lower is better.

Method	Kodak-24	DIV2K
Gaussifier	**0.0331**	**0.0143**
Gaussifier-20k	0.0676	0.0751
AIR	0.0682	0.0987
Instant-GI	0.1594	0.1216
Fast-2DGS	0.3245	0.3119

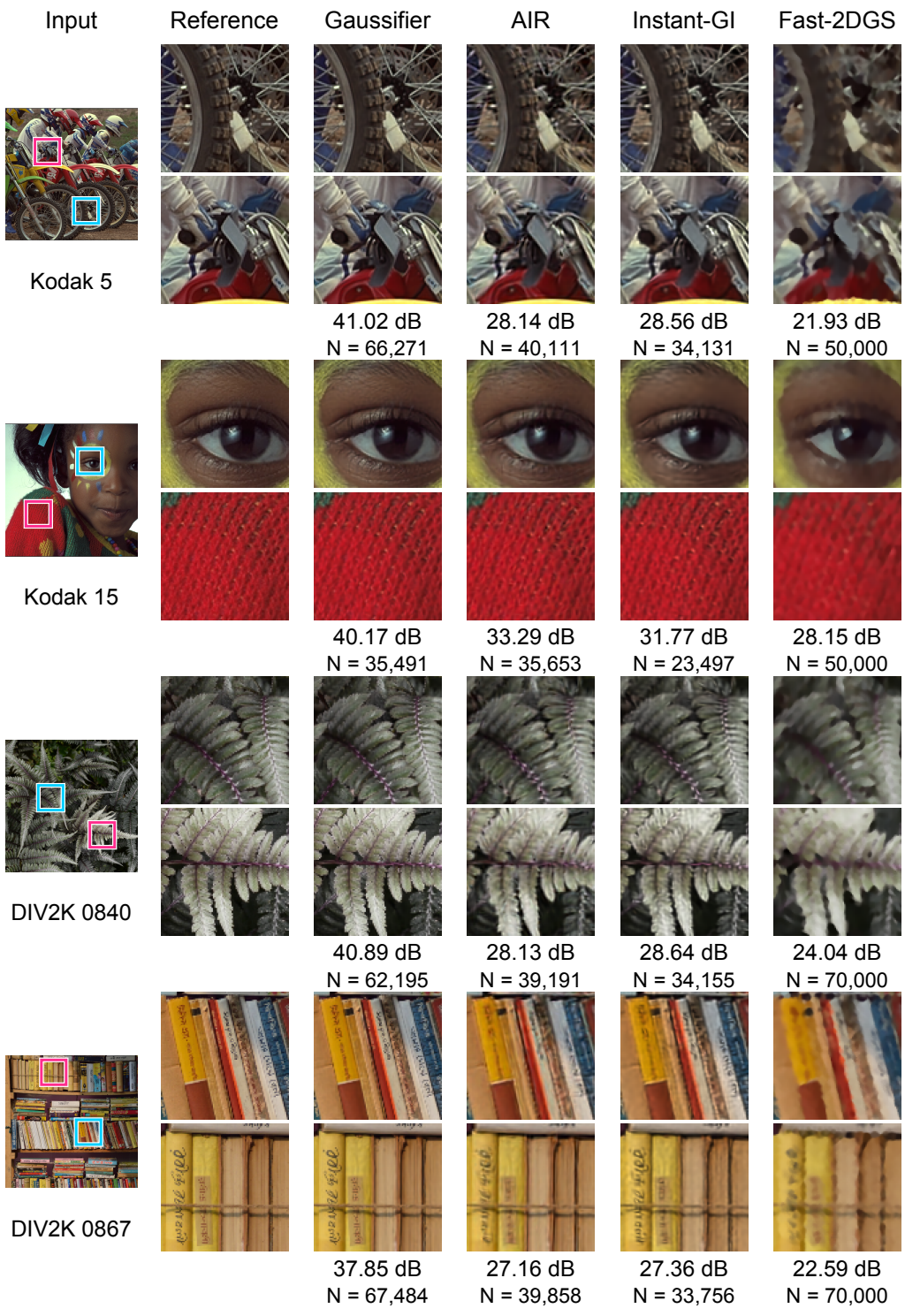

Figure 5: Additional feedforward reconstruction results. All models use the default settings of Table 1, with released baseline checkpoints and no per-image fitting. Each example shows the input and two corresponding 96×96 crops. Gaussifier uses its predicted count and six refinement updates. AIR uses ps5, and Fast-2DGS uses 50k/70k Gaussians on Kodak-24/DIV2K. PSNR and N refer to full images. Fast-2DGS is displayed with seed 0 and its corresponding PSNR.

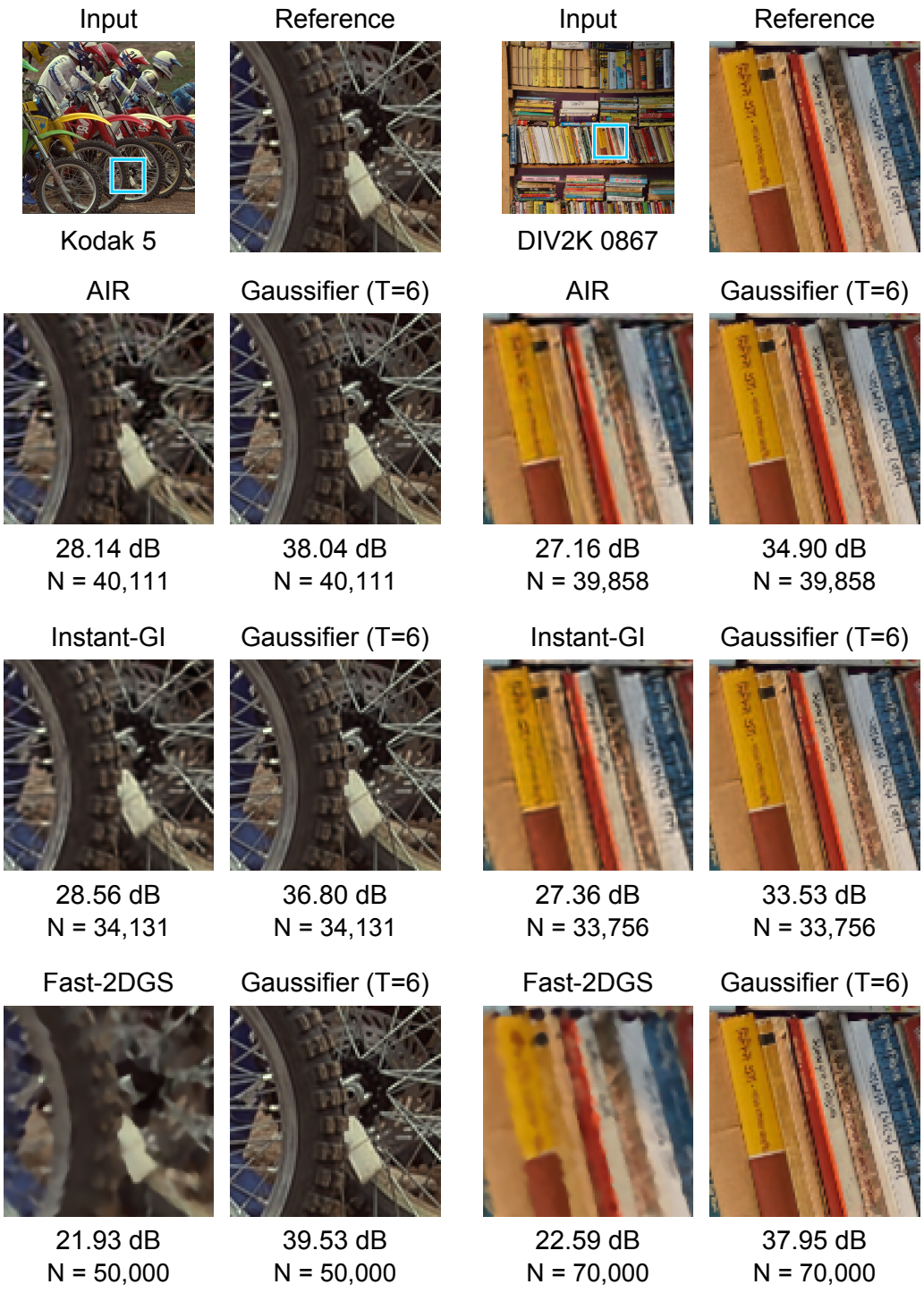

Figure 6: Reconstruction quality at matched Gaussian counts. The input and reference crop are shown once above each group of comparisons. For each image and baseline, Gaussifier uses the same Gaussian count and six refinement updates ($T = 6$). Baselines use their released checkpoints, with AIR using ps5, and Gaussifier reconstructions are produced by its TensorRT/AOTInductor deployment. PSNR and N refer to the full image. Fast-2DGS is displayed with seed 0 and its corresponding PSNR.

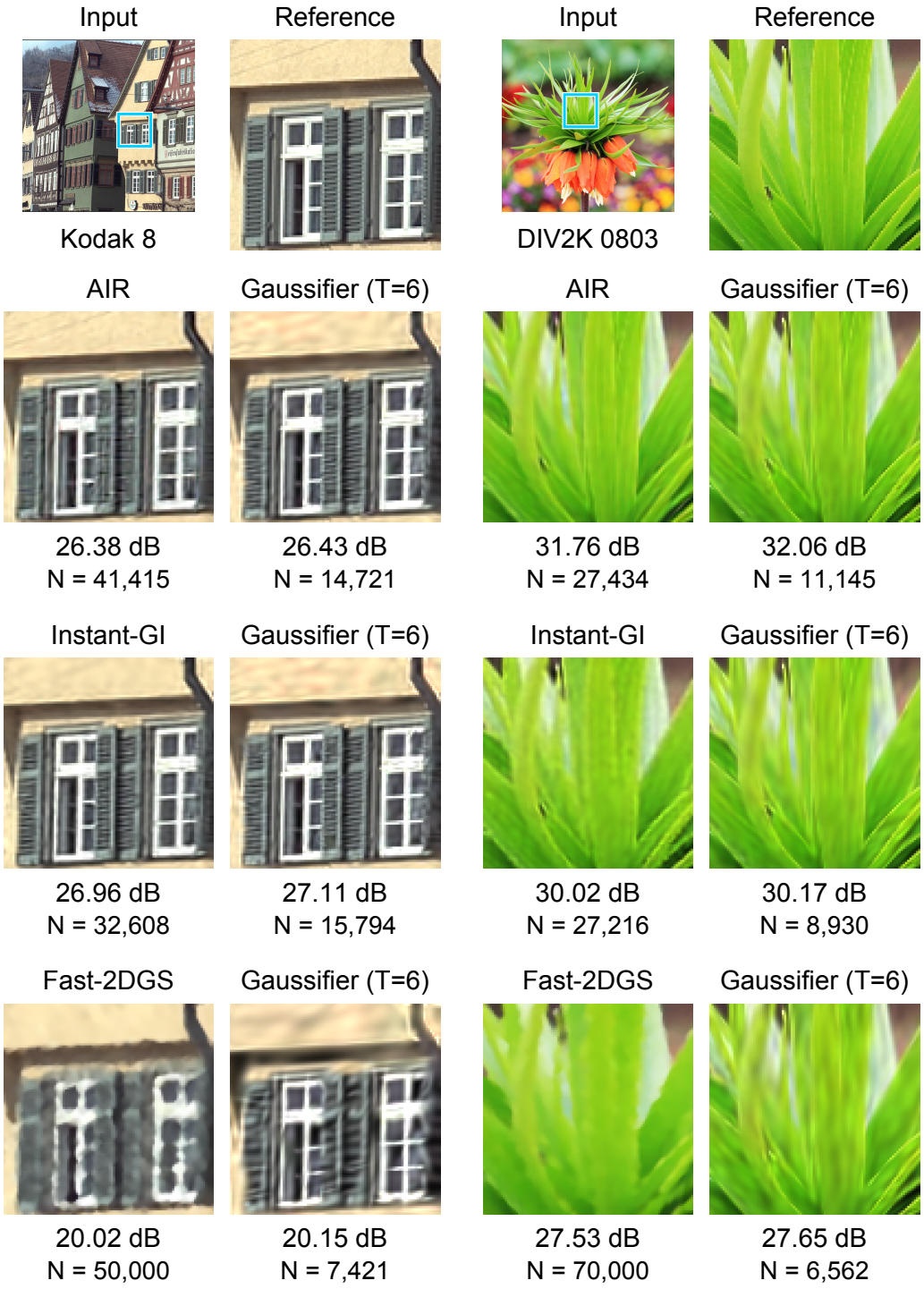

Figure 7: Gaussian counts at baseline reconstruction quality. The input and reference crop are shown once above each group of comparisons. For each image and baseline, we select the smallest evaluated Gaussian count whose full-image PSNR reaches the baseline target. Gaussifier uses six refinement updates ($T = 6$) for every comparison, with reconstructions produced by its TensorRT/AOTInductor deployment. AIR uses the ps5 checkpoint. The Fast-2DGS target averages the PSNR of seeds 0–2, while its displayed reconstruction and PSNR use seed 0. PSNR and N refer to full images.