Supplemental Material:
Analyzing and Improving the Training Dynamics of Diffusion Models

A. Additional results

A.1. Generated images

Figure 7 shows hand-selected images generated using our largest (XXL) ImageNet-512 model without classifier-free guidance. Figures 25–27 show uncurated images from the same model for various ImageNet classes, with guidance strength selected per class.

A.2. Quality vs. compute

Figure 1 in the main paper quantifies the model's cost using gigaflops per evaluation, but this is just one possible option. We could equally well consider several alternative definitions for the model's cost.

Figure 8 shows that the efficiency improvement observed in Figure 1 is retained when the model's cost is quantified using the number of trainable parameters instead. Figure 9 plots the same with respect to the sampling cost per image, demonstrating even greater improvements due to our low number of score function evaluations (NFE). Finally, Figure 10 plots the training cost of the model. According to all of these metrics, our model reaches the same quality much quicker, and proceeds to improve the achievable result quality significantly.

Figure 11a shows the convergence of our different configurations as a function of wall clock time; the early cleanup done in CONFIG B improves both convergence and execution speed in addition to providing a cleaner starting point for experimentation. During the project we tracked various quality metrics in addition to FID, including Inception score [37], KID [4] and Recall [21]. We standardized to FID because of its popularity and because the other metrics were largely consistent with it — see Figure 11b,c compared to Figure 5a.

Figure 12 shows post-hoc EMA sweeps for a set of snapshots for our XXL-sized ImageNet-512 model with and without dropout. We observe that in this large model, overfitting starts to compromise the results without dropout, while a 10% dropout allows steady convergence. Figure 10 further shows the convergence of different model sizes as a function of training cost with and without dropout. For the smaller models (XS, S) dropout is detrimental, but for the larger models it clearly helps, albeit at a cost of slightly slower initial convergence.

Unconditional model	FID ↓	Total capacity (Gparams)	Sampling cost (Tflops)	EMA length
XS	1.81	1.65	38.9	1.5%
S	1.80	1.80	42.5	1.5%
M	1.80	2.02	47.4	1.5%
L	1.86	2.30	53.8	2.0%
XL	1.82	2.64	61.6	2.0%
XXL	1.85	3.05	70.8	2.0%

Table 4. Effect of the unconditional model's size in guiding our XXL-sized ImageNet-512 model. The total capacity and sampling cost refer to the combined cost of the XXL-sized conditional model and the chosen unconditional model. Guidance strength of 1.2 was used in this test.

A.3. Guidance vs. unconditional model capacity

Table 4 shows quantitatively that using a large unconditional model is not useful in classifier-free guidance. Using a very small unconditional model for guiding the conditional model reduces the computational cost of guided diffusion by almost 50%. The EMA lengths in the table apply to both conditional and unconditional model; it is typical that very short EMAs yield best results when sampling with guidance.

A.4. Learning rate vs. EMA length

Figure 13 visualizes the interaction between EMA length and learning rate. While a sweet spot for the learning rate decay parameter still exists ($t_{\text{ref}} = 70\text{k}$ in this case), the possibility of sweeping over the EMA lengths post hoc drastically reduces the importance of this exact choice. A wide bracket of learning rate decays $t_{\text{ref}} \in [30\text{k}, 160\text{k}]$ yields FIDs within 10% of the optimum using post-hoc EMA.

In contrast, if the EMA length was fixed at 13%, varying t_{ref} would increase FID much more, at worst by 72% in the tested range.

A.5. Fréchet distances using DINOv2

The DINOv2 feature space [29] has been observed to align much better with human preferences compared to the widely used InceptionV3 feature space [38]. We provide a version of Table 2 using the Fréchet distance computed in the DINOv2 space ($\text{FD}_{\text{DINOv2}}$) in Table 5 to facilitate future comparisons.

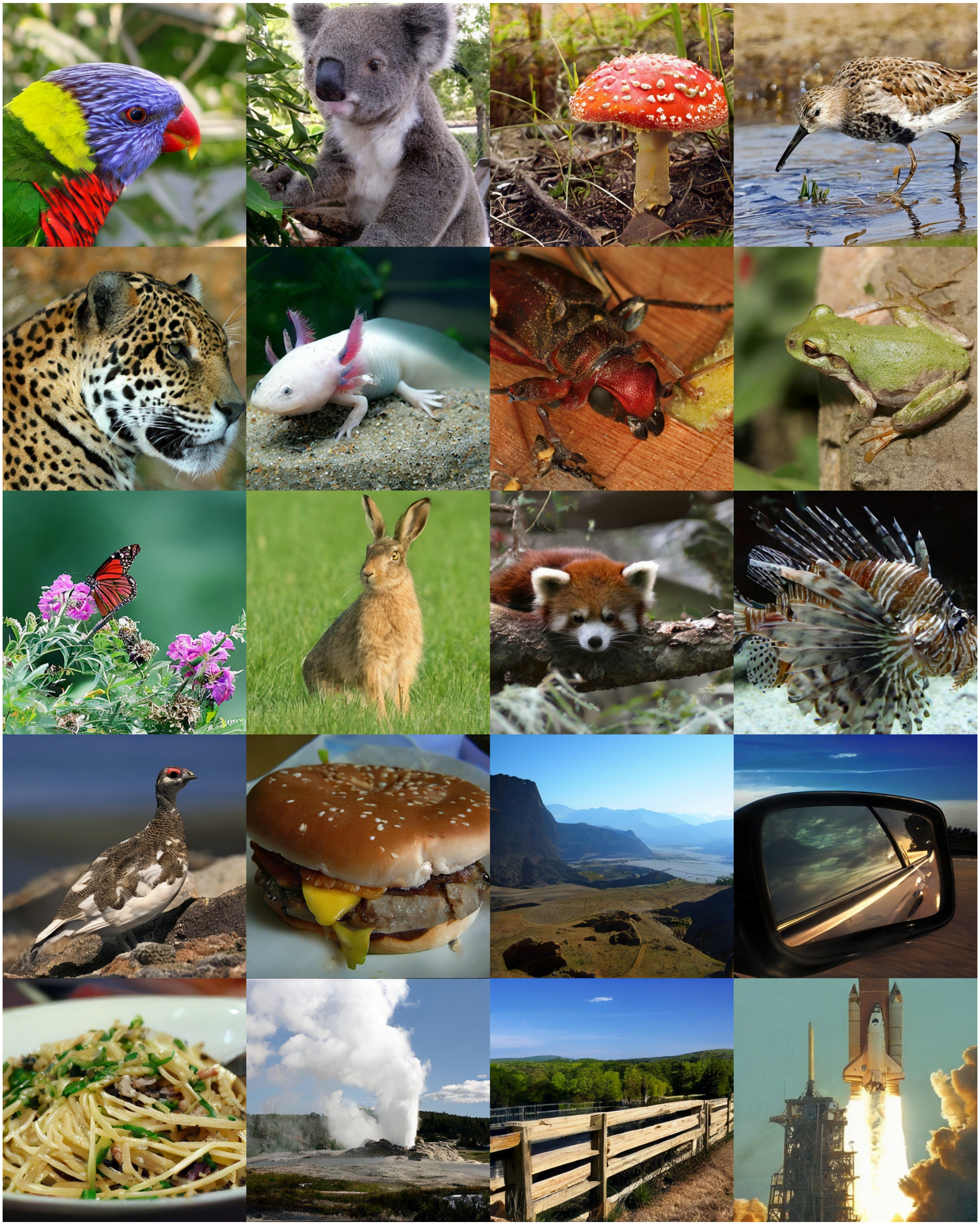


Figure 7. Selected images generated using our largest (XXL) ImageNet-512 model without guidance.

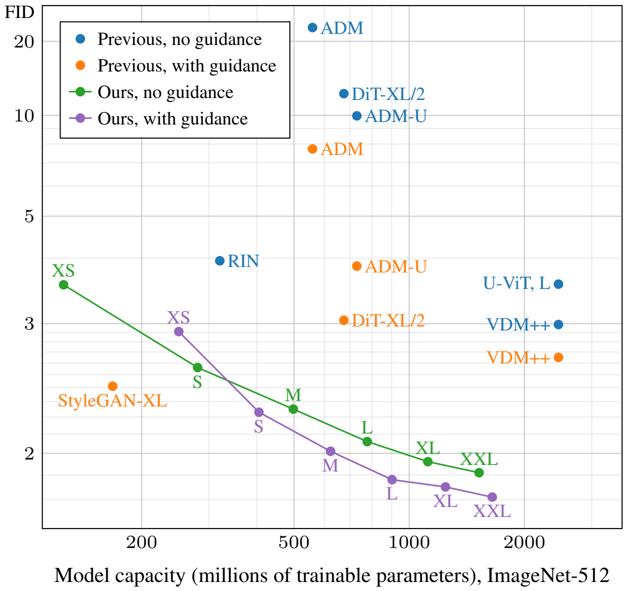


Figure 8. FID vs. model capacity on ImageNet-512. For our method with guidance, we account for the number of parameters in the XS-sized unconditional model.

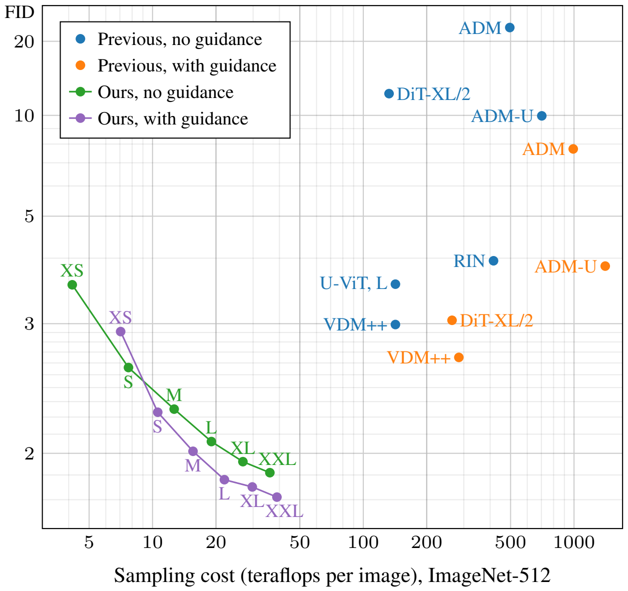


Figure 9. FID vs. sampling cost on ImageNet-512. For latent diffusion models (DiT-XL/2 and ours), we include the cost of running the VAE decoder at the end (1260.9 gigaflops per image).

We use the publicly available implementation[2] by Stein et al. [38] for computing $\mathrm{FD}_{\mathrm{DINOv2}}$. We use 50,000 generated images and all 1,281,167 available real images, following the established best practices in FID computation. Class labels for the 50k generated samples are drawn from a uniform distribution. We evaluate FD only once per 50k sample as we observe little random variation between runs.

Figure 14 compares FID and $\mathrm{FD}_{\mathrm{DINOv2}}$ as a function of EMA length. We can make three interesting observations. First, without guidance, the optima of the two CONFIG G curves (green) are in a clearly different place, with $\mathrm{FD}_{\mathrm{DINOv2}}$ preferring longer EMA. The disagreement between the two metrics is quite significant: FID considers $\mathrm{FD}_{\mathrm{DINOv2}}$'s optimum (19%) to be a poor choice, and vice versa.

Second, with guidance strength 1.4 (the optimal choice for FID according to Figure 6) the curves are astonishingly different. While both metrics agree that a modest amount of guidance is helpful, their preferred EMA lengths are totally different (2% vs 14%). FID considers $\mathrm{FD}_{\mathrm{DINOv2}}$'s optimum (14%) to be a terrible choice and vice versa. Based on a cursory assessment of the generated images, it seems that $\mathrm{FD}_{\mathrm{DINOv2}}$ prefers images with better global coherency, which often maps to higher perceptual quality, corroborating the conclusions of Stein et al. [38]. The significant differences in the optimal EMA length highlight the importance of searching the optimum specifically for the chosen quality metric.

Third, $\mathrm{FD}_{\mathrm{DINOv2}}$ prefers higher guidance strength than FID (1.9 vs 1.4). FID considers 1.9 clearly excessive.

The figure furthermore shows that our changes (CONFIG B vs G) yield an improvement in $\mathrm{FD}_{\mathrm{DINOv2}}$ that is at least as significant as the drop we observed using FID.

A.6. Activation and weight magnitudes

Figure 15 shows an extended version of Figure 3, including activation and weight magnitude plots for CONFIG B–G measured using both max and mean aggregation over each resolution bucket. The details of the computation are as follows.

We first identify all trainable weight tensors within the U-Net encoder/decoder blocks of each resolution, including those in the associated self-attention layers. This yields a set of tensors for each of the eight resolution buckets identified in the legend, i.e., $\{\mathrm{Enc}, \mathrm{Dec}\}\times\{8\times 8, \ldots, 64\times 64\}$. The analyzed activations are the immediate outputs of the operations involving these tensors before any nonlinearity, and the analyzed weights are the tensors themselves.

In CONFIG B, we do not include trainable biases in the weight analysis, but the activations are measured after applying the biases. In CONFIG G, we exclude the learned scalar gains from the weight analysis, but measure the activations after the gains have been applied.

Activations. The activation magnitudes are computed as an expectation over 4096 training samples. Ignoring the minibatch axis for clarity, most activations are shaped $N\times H\times W$ where N is the number of feature maps and H and W are the spatial dimensions. For the purposes of analysis, we reshape these to $N\times M$ where $M = HW$. The

[2] `https://github.com/layer6ai-labs/dgm-eval`

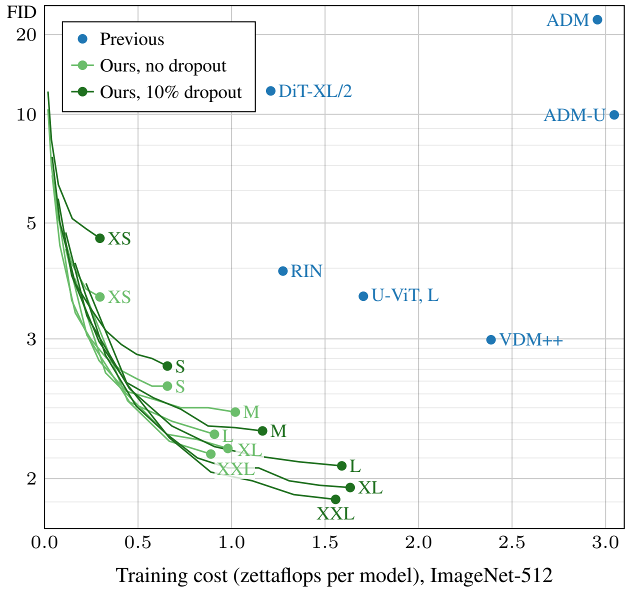


Figure 10. FID vs. training cost on ImageNet-512 without guidance. Note that one zettaflop $= 10^{21}$ flops $= 10^{12}$ gigaflops. We assume that one training iteration is three times as expensive as evaluating the model (i.e., forward pass, backprop to inputs, backprop to weights).

outputs of the linear transformation of the class embedding vector are considered to have shape $N \times 1$.

Given a potentially reshaped activation tensor $\mathbf{h} \in \mathbb{R}^{N \times M}$, we compute the magnitudes of the individual features $\mathbf{h}_i$ as

$$\mathcal{M}[\mathbf{h}_i] \;=\; \sqrt{\frac{1}{M}\sum_{j=1}^{M}\mathbf{h}_{i,j}^2}. \tag{3}$$

The result contains the per-feature L_2 norms of the activations in tensor $\mathbf{h}$, scaled such that unit-normal distributed activations yield an expected magnitude of 1 regardless of their dimensions.

All of these per-feature scalar magnitudes within a resolution bucket are aggregated into a single number by taking either their maximum or their mean. Taking the maximum magnitude (Figure 3 and Figure 15, left half) ensures that potential extreme behavior is not missed, whereas the mean magnitude (Figure 15, right half) is a closer indicator of average behavior. Regardless of the choice, the qualitative behavior is similar.

Weights. All weight tensors under analysis are of shape $N \times \cdots$ where N is the number of output features. We thus reshape them all into $N \times M$ and compute the per-output-feature magnitudes using Equation 3. Similar to activations, this ensures that unit-normal distributed weights have an expected magnitude of 1 regardless of degree or dimensions of the weight tensor. We again aggregate all magnitudes within a resolution bucket into a single number by taking either the maximum or the mean. Figure 3 displays maximum magnitudes, whereas the extended version in Figure 15 shows both maximum and mean magnitudes.

ImageNet-512	**$\text{FD}_{\text{DINOv2}}$ ↓**		**Model size**		
	no CFG	w/CFG	Mparams	Gflops	NFE
EDM2-XS	103.39	79.94	125	46	63
EDM2-S	68.64	52.32	280	102	63
EDM2-M	58.44	41.98	498	181	63
EDM2-L	52.25	38.20	777	282	63
EDM2-XL	45.96	35.67	1119	406	63
EDM2-XXL	42.84	33.09	1523	552	63

Table 5. Version of Table 2 using $\text{FD}_{\text{DINOv2}}$ instead of FID on ImageNet-512. The "w/CFG" and "no CFG" columns show the lowest FID obtained with and without classifier-free guidance, respectively. NFE tells how many times the score function is evaluated when generating an image.

B. Architecture details

In this section, we present comprehensive details for the architectural changes introduced in Section 2. Figures 16–22 illustrate the architecture diagram corresponding to each configuration, along with the associated hyperparameters. In order to observe the individual changes, we invite the reader to flip through the figures in digital form.

B.1. EDM baseline (CONFIG A)

Our baseline corresponds to the ADM [7] network as implemented in the EDM [17] framework, operating in the latent space of a pre-trained variational autoencoder (VAE) [33]. We train the network for 2^{19} training iterations with batch size 4096, i.e., 2147.5 million images, using the same hyperparameter choices that were previously used for ImageNet-64 by Karras et al. [17]. In this configuration, we use traditional EMA with a half-life of 50M images, i.e., 12k training iterations, which translates to $\sigma_{\text{rel}} \approx 0.034$ at the end of the training. The architecture and hyperparameters as summarized in Figure 16.

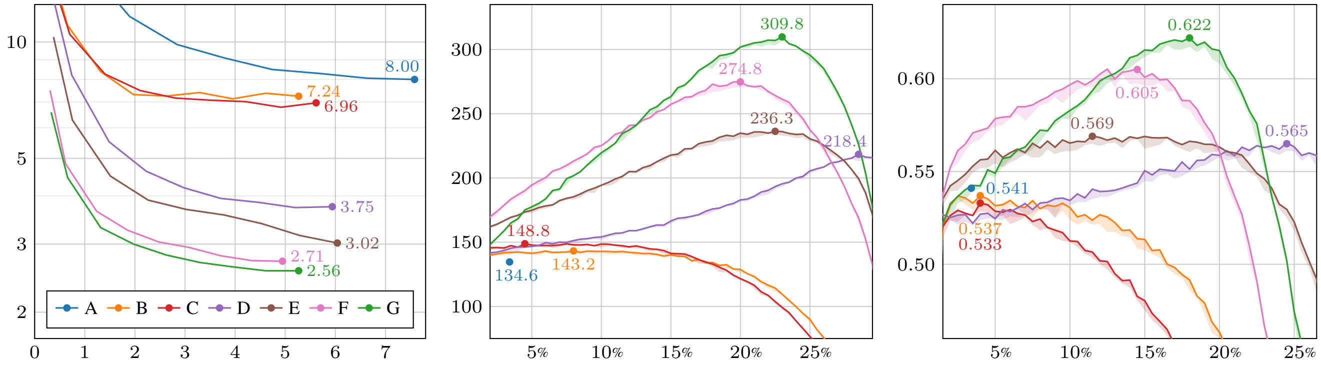
Preconditioning. Following the EDM framework, the network implements denoiser $\hat{\boldsymbol{y}} = D_\theta(\boldsymbol{x}; \sigma, \boldsymbol{c})$, where $\boldsymbol{x}$ is a noisy input image, σ is the corresponding noise level, $\boldsymbol{c}$ is a one-hot class label, and $\hat{\boldsymbol{y}}$ is the resulting denoised image; in the following, we will omit $\boldsymbol{c}$ for conciseness. The framework further breaks down the denoiser as

$$D_\theta(\boldsymbol{x};\sigma) \;=\; c_{\text{skip}}(\sigma)\boldsymbol{x} + c_{\text{out}}(\sigma)F_\theta\big(c_{\text{in}}(\sigma)\boldsymbol{x}; c_{\text{noise}}(\sigma)\big) \tag{4}$$

$$c_{\text{skip}}(\sigma) \;=\; \sigma_{\text{data}}^2 \,\big/\, \big(\sigma^2 + \sigma_{\text{data}}^2\big) \tag{5}$$

$$c_{\text{out}}(\sigma) \;=\; (\sigma \cdot \sigma_{\text{data}}) \,\big/\, \sqrt{\sigma^2 + \sigma_{\text{data}}^2} \tag{6}$$

$$c_{\text{in}}(\sigma) \;=\; 1 \,\big/\, \sqrt{\sigma^2 + \sigma_{\text{data}}^2} \tag{7}$$



(a) FID ↓ vs. training time (days) (b) Inception score ↑ vs. EMA length (c) Recall ↑ vs. EMA length

Figure 11. CONFIG A–G on ImageNet-512. **(a)** Convergence in wall-clock time for equal-length training runs. **(b, c)** Additional metrics.

$$c_{\text{noise}}(\sigma) \;=\; \tfrac{1}{4}\ln(\sigma), \tag{8}$$

where the inputs and outputs of the raw network layers F_θ are preconditioned according to c_{in}, c_{out}, c_{skip}, and c_{noise}. σ_{data} is the expected standard deviation of the training data. The preconditioning is reflected in Figure 16 by the blue boxes around the main inputs and outputs.

Latent diffusion. For ImageNet-512, we follow Peebles and Xie [30] by preprocessing each 512×512×3 image in the dataset with a pre-trained off-the-shelf VAE encoder from Stable Diffusion[3] and postprocessing each generated image with the correspoding decoder. For a given input image, the encoder produces a 4-channel latent at 8×8 times lower resolution than the original, yielding a dimension of 64×64×4 for $\boldsymbol{x}$ and $\hat{\boldsymbol{y}}$. The mapping between images and latents is not strictly bijective: The encoder turns a given image into a distribution of latents, where each channel c of each pixel (x, y) is drawn independently from $\mathcal{N}(\mu_{x,y,c}, \sigma^2_{x,y,c})$. When preprocessing the dataset, we store the values of $\mu_{x,y,c}$ and $\sigma_{x,y,c}$ as 32-bit floating point, and draw a novel sample each time we encounter a given image during training.

The EDM formulation in Equation 4 makes relatively strong assumptions about the mean and standard deviation of the training data. We choose to normalize the training data globally to satisfy these assumptions — as opposed to, e.g., changing the value of σ_{data}, which might have far-reaching consequences in terms of the other hyperparameters. We thus keep σ_{data} at its default value 0.5, subtract $[5.81, 3.25, 0.12, -2.15]$ from the latents during dataset preprocessing to make them zero mean, and multiply them by $0.5 \;/\; [4.17, 4.62, 3.71, 3.28]$ to make their standard deviation agree with σ_{data}. When generating images, we undo this normalization before running the VAE decoder.

Architecture walkthrough. The ADM [7] network starts by feeding the noisy input image, multiplied by c_{noise}, through an input block ("In") to expand it to 192 channels. It then processes the resulting activation tensor through a series of encoder and decoder blocks, organized as a U-Net structure [34] and connected to each other via skip connections (faint curved arrows). At the end, the activation tensor is contracted back to 4 channels by an output block ("Out"), and the final denoised image is obtained using c_{out} and c_{skip} as defined by Equation 4. The encoder gradually decreases the resolution from 64×64 to 32×32, 16×16, and 8×8 by a set of downsampling blocks ("EncD"), and the channel count is simultaneously increased from 192 to 384, 576, and 768. The decoder implements the same progression in reverse using corresponding upsampling blocks ("DecU").

The operation of the encoder and decoder blocks is conditioned by a 768-dimensional embedding vector, obtained by feeding the noise level σ and class label $\boldsymbol{c}$ through a separate embedding network ("Embedding"). The value of $c_{\text{noise}}(\sigma)$ is fed through a sinusoidal timestep embedding layer[4,5]("PosEmb") to turn it into a 192-dimensional feature vector. The result is then processed by two fully-connected layers with SiLU nonlinearity [11], defined as

$$\operatorname{silu}(x) \;=\; \frac{x}{1+e^{-x}}, \tag{9}$$

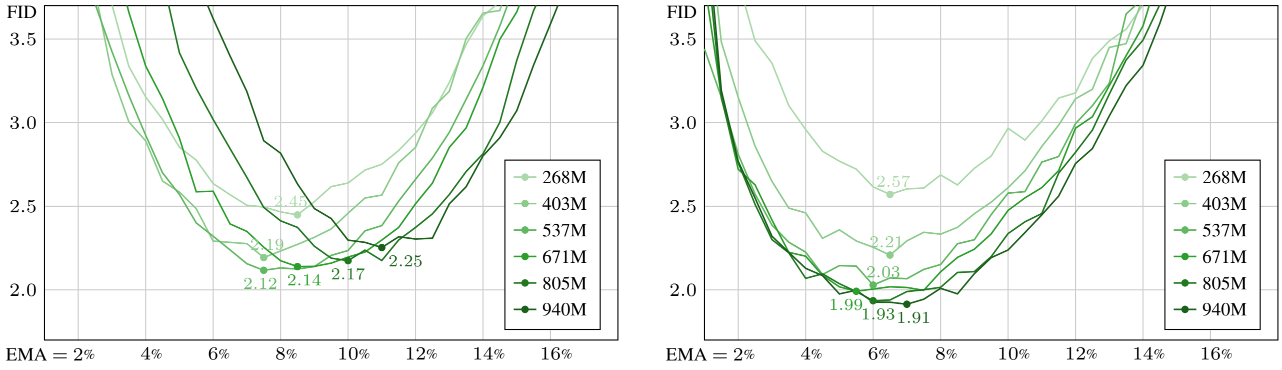
adding in a learned per-class embedding before the second nonlinearity.

The encoder and decoder blocks follow the standard pre-activation design of ResNets [10]. The main path (bold line) undergoes minimal processing: It includes an optional 2×2 upsampling or downsampling using box filter if the resolution changes, and an 1×1 convolution if the number of channels changes. The residual path employs two 3×3 convolutions, preceded by group normalization and SiLU

[3] `https://huggingface.co/stabilityai/sd-vae-ft-mse`

[4] `https://github.com/openai/guided-diffusion/blob/22e0df8183507e13a7813f8d38d51b072ca1e67c/guided_diffusion/nn.py#L103`

[5] `https://github.com/NVlabs/edm/blob/62072d2612c7da05165d6233d13d17d71f213fee/training/networks.py#L193`



(a) EDM2-XXL, no dropout

(b) EDM2-XXL, 10% dropout

Figure 12. Effect of dropout on the training of EDM2-XXL in ImageNet-512. **(a)** Without dropout, the training starts to overfit after 537 million training images. **(b)** With dropout, the training starts off slightly slower, but it makes forward progress for much longer.

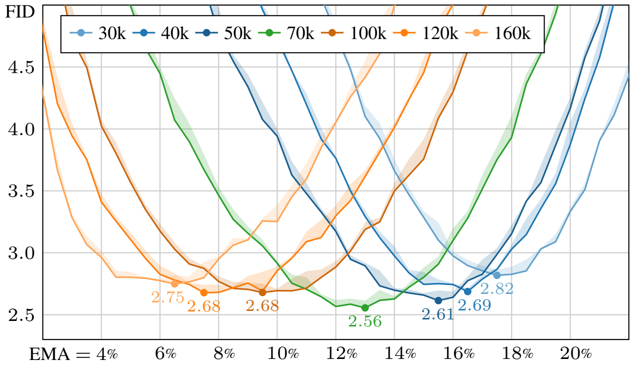


Figure 13. Interaction between EMA length and learning rate decay (t_{ref}, different colors) using EDM2-S on ImageNet-512.

nonlinearity. The group normalization computes empirical statistics for each group of 32 channels, normalizes them to zero mean and unit variance, and then applies learned per-group scaling and bias. Between the convolutions, each channel is further scaled and biased based on the value of the embedding vector, processed by a per-block fully-connected layer. The ADM architecture further employs dropout before the second convolution, setting individual elements of the activation tensor to zero with 10% probability during training. The U-Net skip connections originate from the outputs of the encoder blocks, and they are concatenated to the inputs of the corresponding decoder blocks.

Most of the encoder and decoder blocks operating at 32×32 resolution and below ("EncA" and "DecA") further employ self-attention after the residual branch. The implementation follows the standard multi-head scaled dot product attention [41], where each pixel of the incoming activation tensor is treated as a separate token. For a single attention head, the operation is defined as

$$\mathbf{A} \;=\; \operatorname{softmax}(\mathbf{W})\mathbf{V} \qquad (10)$$

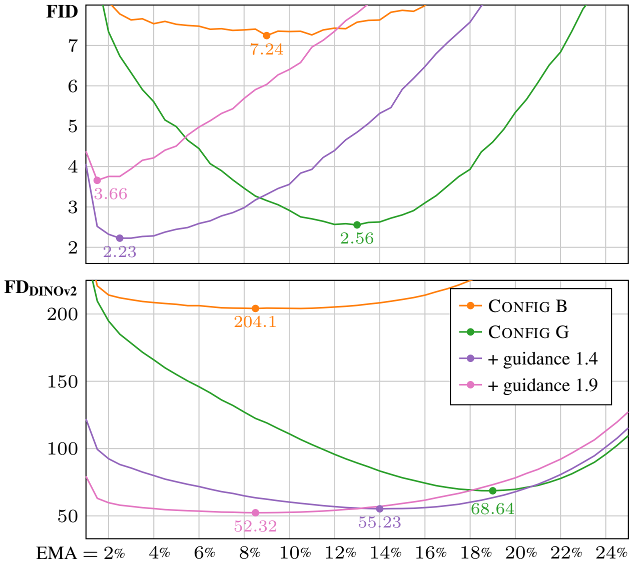


Figure 14. FID and $\text{FD}_{\text{DINOv2}}$ as a function of EMA length using S-sized models on ImageNet-512. Configs B and G illustrate the improvement from our changes. We also show two guidance strengths: FID's optimum (1.4) and $\text{FD}_{\text{DINOv2}}$'s optimum (1.9).

$$\mathbf{W} \;=\; \frac{1}{\sqrt{N_c}}\mathbf{Q}\mathbf{K}^\top, \qquad (11)$$

where $\mathbf{Q} = [\mathbf{q}_1, \ldots]^\top$, $\mathbf{K} = [\mathbf{k}_1, \ldots]^\top$, and $\mathbf{V} = [\mathbf{v}_1, \ldots]^\top$ are matrices containing the query, key, and value vectors for each token, derived from the incoming activations using a 1×1 convolution. The dimensionality of the query and key vectors is denoted by N_c.

The elements of the weight matrix in Equation 11 can equivalently be expressed as dot products between the individual query and key vectors:

$$w_{i,j} \;=\; \frac{1}{\sqrt{N_c}}\big\langle \mathbf{q}_i, \mathbf{k}_j \big\rangle. \qquad (12)$$

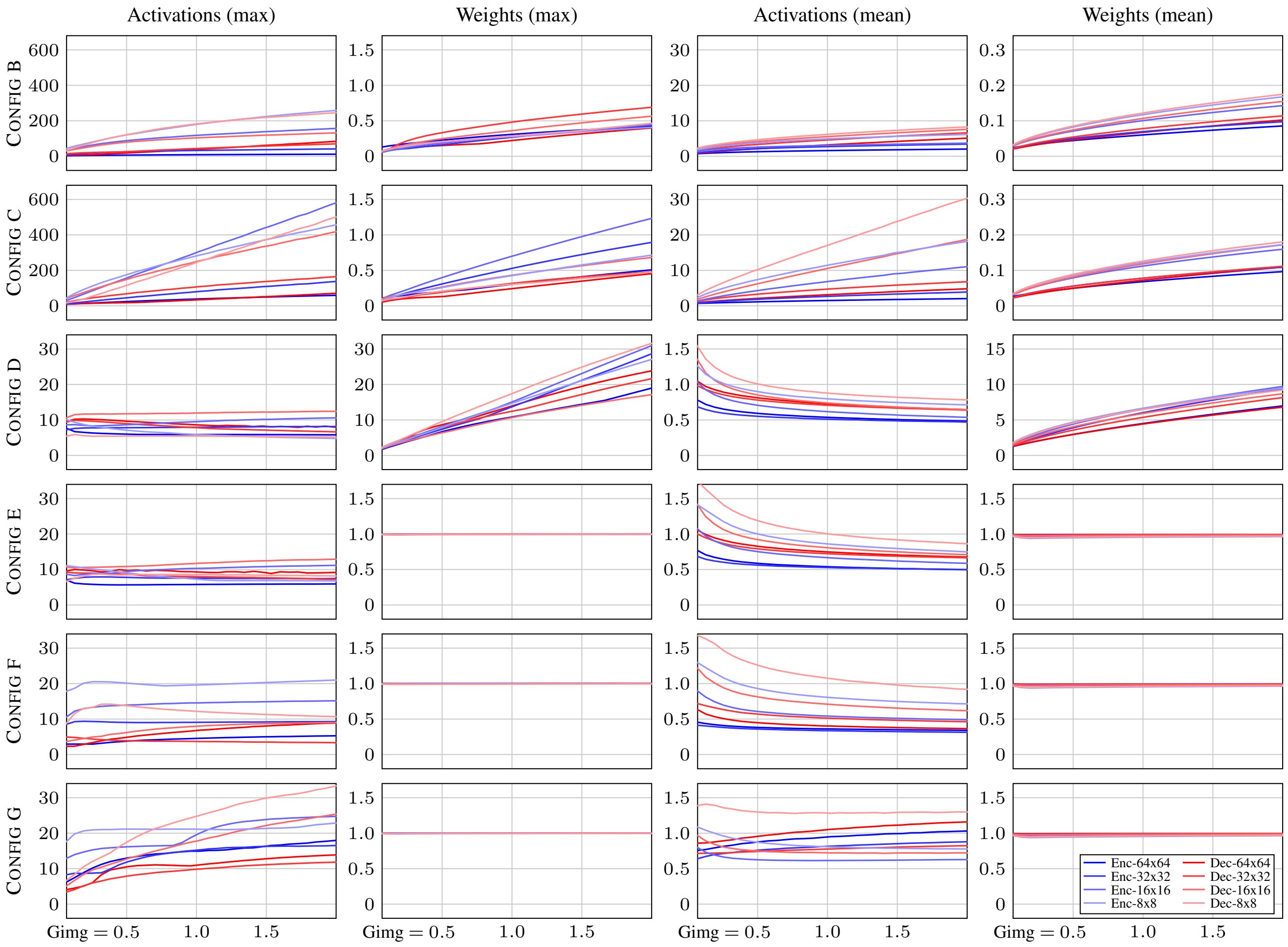


Figure 15. Training-time evolution of the maximum and mean dimension-weighted L_2 norms of activations and weights over different depths of the the EMA-averaged score network. As discussed in Section 2, our architectural modifications aim to standardize the activation magnitudes in CONFIG D and weight magnitudes in CONFIG E. Details of the computation are discussed in Appendix A.6.

Equation 10 is repeated for each attention head, after which the resulting tokens $\mathbf{A}$ are concatenated, transformed by a 1×1 convolution, and added back to the main path. The number of heads N_h is determined by the incoming channel count so that there is one head for each set of 64 channels. The dot product and softmax operations are executed using 32-bit floating point to avoid overflows, even though the rest of the network uses 16-bit floating point.

The weights of almost every convolution and fully-connected layer are initialized using He's uniform init [9], and the corresponding biases are also drawn from the same distribution. There are two exceptions, however: The per-class embedding vectors are initialized to $\mathcal{N}(0,1)$, and the weights and biases of the last convolution of the residual blocks, self-attention blocks, and the final output block are initialized to zero (dark green). This has the effect that $D_\theta(\boldsymbol{x},\sigma) = c_{\text{skip}}(\sigma)\,\boldsymbol{x}$ after initialization.

Training loss. Following EDM [17], the denoising score matching loss for denoiser D_θ on noise level σ is given by

$$\mathcal{L}(D_\theta;\sigma) \;=\; \mathbb{E}_{\boldsymbol{y},\boldsymbol{n}}\Big[\big\|D_\theta(\boldsymbol{y}+\boldsymbol{n};\sigma)-\boldsymbol{y}\big\|_2^2\Big], \tag{13}$$

where $\boldsymbol{y}\sim p_{\text{data}}$ is a clean image sampled from the training set and $\boldsymbol{n}\sim\mathcal{N}\big(\mathbf{0},\sigma^2\mathbf{I}\big)$ is i.i.d. Gaussian noise.

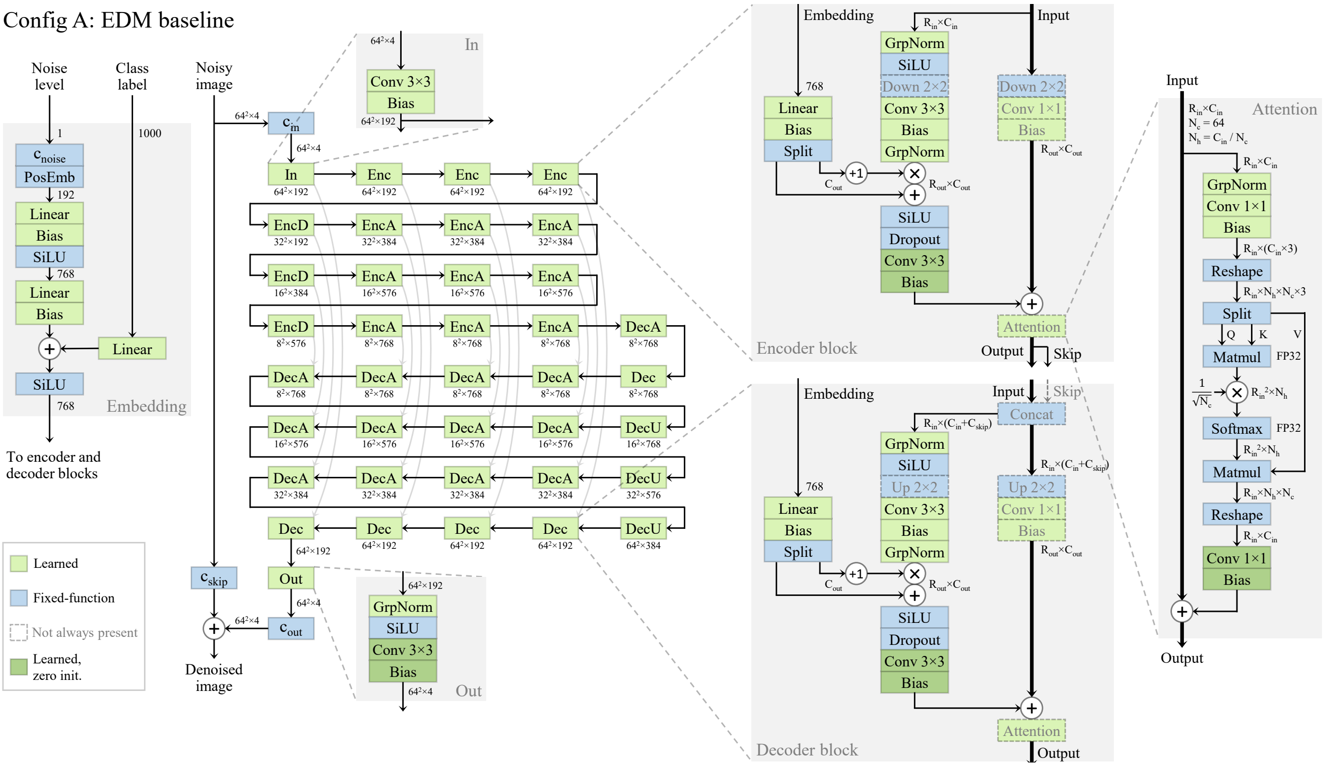
The overall training loss is defined [17] as a weighted expectation of $\mathcal{L}(D_\theta;\sigma)$ over the noise levels:

$$\mathcal{L}(D_\theta) \;=\; \mathbb{E}_\sigma\big[\lambda(\sigma)\mathcal{L}(D_\theta;\sigma)\big] \tag{14}$$

$$\lambda(\sigma) \;=\; \big(\sigma^2+\sigma_{\text{data}}^2\big)\,/\,(\sigma\cdot\sigma_{\text{data}})^2 \tag{15}$$

$$\ln(\sigma) \;\sim\; \mathcal{N}\big(P_{\text{mean}},P_{\text{std}}^2\big), \tag{16}$$

where the distribution of noise levels is controlled by hyperparameters P_{mean} and P_{std}. The weighting function $\lambda(\sigma)$ in Equation 15 ensures that $\lambda(\sigma)\mathcal{L}(D_\theta;\sigma)=1$ at the beginning of the training, effectively equalizing the contribution of each noise level with respect to $\nabla_\theta\mathcal{L}(D_\theta)$.



Number of GPUs	32	Learning rate max (α_{ref})	0.0001	Adam β_1	0.9	FID	8.00
Minibatch size	4096	Learning rate decay (t_{ref})	∞	Adam β_2	0.999	EMA length (σ_{rel})	0.034
Duration (Mimg)	2147.5	Learning rate rampup (Mimg)	10	Loss scaling	100	Model capacity (Mparams)	295.9
Mixed-precision (FP16)	partial	Noise distribution mean (P_{mean})	-1.2	Attention res.	32, 16, 8	Model complexity (Gflops)	110.4
Dropout probability	10%	Noise distribution std. (P_{std})	1.2	Attention blocks	22	Sampling cost (Tflops)	8.22

Figure 16. Full architecture diagram and hyperparameters for CONFIG A (EDM baseline).

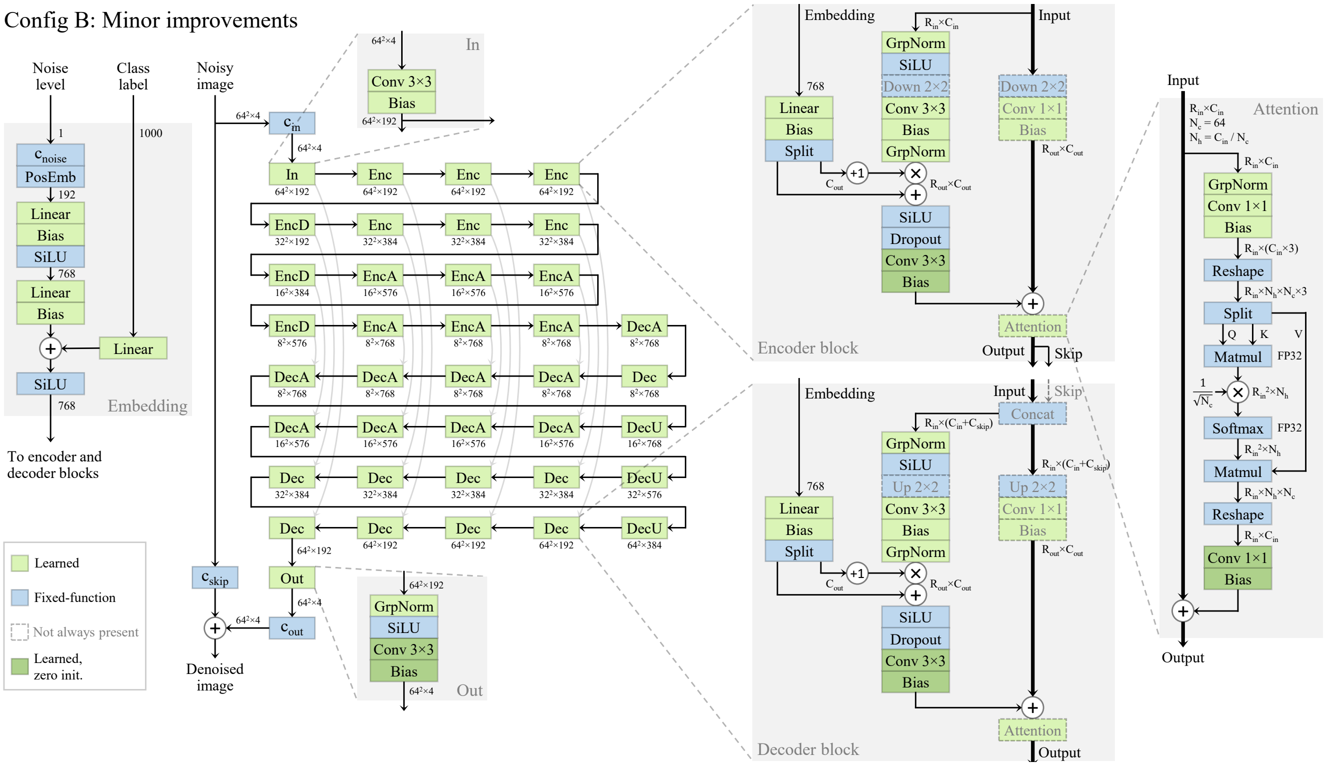
B.2. Minor improvements (CONFIG B)

Since the baseline configuration (CONFIG A) was not originally targeted for latent diffusion, we re-examined the hyperparameter choices to obtain an improved baseline (CONFIG B). Our new hyperparameters are summarized in Figure 17.

In order to speed up convergence, we found it beneficial to halve the batch size (2048 instead of 4096) while doubling the learning rate ($\alpha_{\text{ref}} = 0.0002$ instead of 0.0001), and to significantly reduce Adam's response time to changes in gradient magnitudes ($\beta_2 = 0.99$ instead of 0.999). These changes had the largest impact towards the beginning of the training, where the network reconfigures itself for the task at hand, but they also helped somewhat towards the end. Furthermore, we found the self-attention layers at 32×32 resolution to be somewhat harmful; removing them improved the overall stability while also speeding up the training. In CONFIG B, we also switch from traditional EMA to our power function averaging profile (Section 3.1), with two averages stored per snapshot for high-quality post-hoc reconstruction (Section 3.2).

Loss weighting. With the EDM training loss (Equation 14), the quality of the resulting distribution tends to be quite sensitive to the choice of P_{mean}, P_{std}, and $\lambda(\sigma)$. The role of P_{mean} and P_{std} is to focus the training effort on the most important noise levels, whereas $\lambda(\sigma)$ aims to ensure that the gradients originating from each noise level are roughly of the same magnitude. Referring to Figure 5a of Karras et al. [17], the value of $\mathcal{L}(D_\theta; \sigma)$ behaves somewhat unevenly over the course of training: It remains largely unchanged for the lowest and highest noise levels, but drops quickly for the ones in between. Karras et al. [17] suggest setting P_{mean} and P_{std} so that the resulting log-normal distribution (Equation 16) roughly matches the location of this in-between region. When operating with VAE latents, we have observed that the in-between region has shifted considerably toward higher noise levels compared to RGB images. We thus set $P_{\text{mean}} = -0.4$ and $P_{\text{std}} = 1.0$ instead of -1.2 and 1.2, respectively, to roughly match its location.

While the choice of $\lambda(\sigma)$ defined by Equation 15 is enough to ensure that the gradient magnitudes are balanced at initialization, this is no longer true as the training progresses. To compensate for the changes in $\mathcal{L}(D_\theta; \sigma)$ that



Number of GPUs	32	Learning rate max (α_{ref})	0.0002	Adam β_1	0.9	FID	7.24
Minibatch size	2048	Learning rate decay (t_{ref})	∞	Adam β_2	0.99	EMA length (σ_{rel})	0.090
Duration (Mimg)	2147.5	Learning rate rampup (Mimg)	10	Loss scaling	100	Model capacity (Mparams)	291.8
Mixed-precision (FP16)	partial	Noise distribution mean (P_{mean})	-0.4	Attention res.	16, 8	Model complexity (Gflops)	100.4
Dropout probability	10%	Noise distribution std. (P_{std})	1.0	Attention blocks	15	Sampling cost (Tflops)	7.59

Figure 17. Full architecture diagram and hyperparameters for CONFIG B (Minor improvements).

happen over time, no static choice of $\lambda(\sigma)$ is sufficient — the weighting function must be able to adapt its shape dynamically. To achieve this, we treat the integration over noise levels in $\mathcal{L}(D_\theta)$ as a form of multi-task learning. In the following, we will first summarize the uncertainty-based weighting approach proposed by Kendall et al. [18], defined over a finite number of tasks, and then generalize it over a continuous set of tasks to replace Equation 14.

Uncertainty-based multi-task learning. In a traditional multi-task setting, the model is simultaneously being trained to perform multiple tasks corresponding to loss terms $\{\mathcal{L}_1, \mathcal{L}_2, \ldots\}$. The naive way to define the overall loss is to take a weighted sum over these individual losses, i.e., $\mathcal{L} = \sum_i w_i \mathcal{L}_i$. The outcome of the training, however, tends to be very sensitive to the choice of weights w_i. This choice can become particularly challenging if the balance between the loss terms changes considerably over time. Kendall et al. [18] propose a principled approach for choosing the weights dynamically, based on the idea of treating the model outputs as probability distributions and maximizing the resulting likelihood. For isotropic Gaussians, this boils down to associating each loss term $\mathcal{L}_i$ with an additional trainable parameter $\sigma_i > 0$, i.e., homoscedastic uncertainty, and defining the overall loss as

$$\mathcal{L} = \sum_i \left[\frac{1}{2\sigma_i^2} \mathcal{L}_i + \ln \sigma_i \right] \tag{17}$$

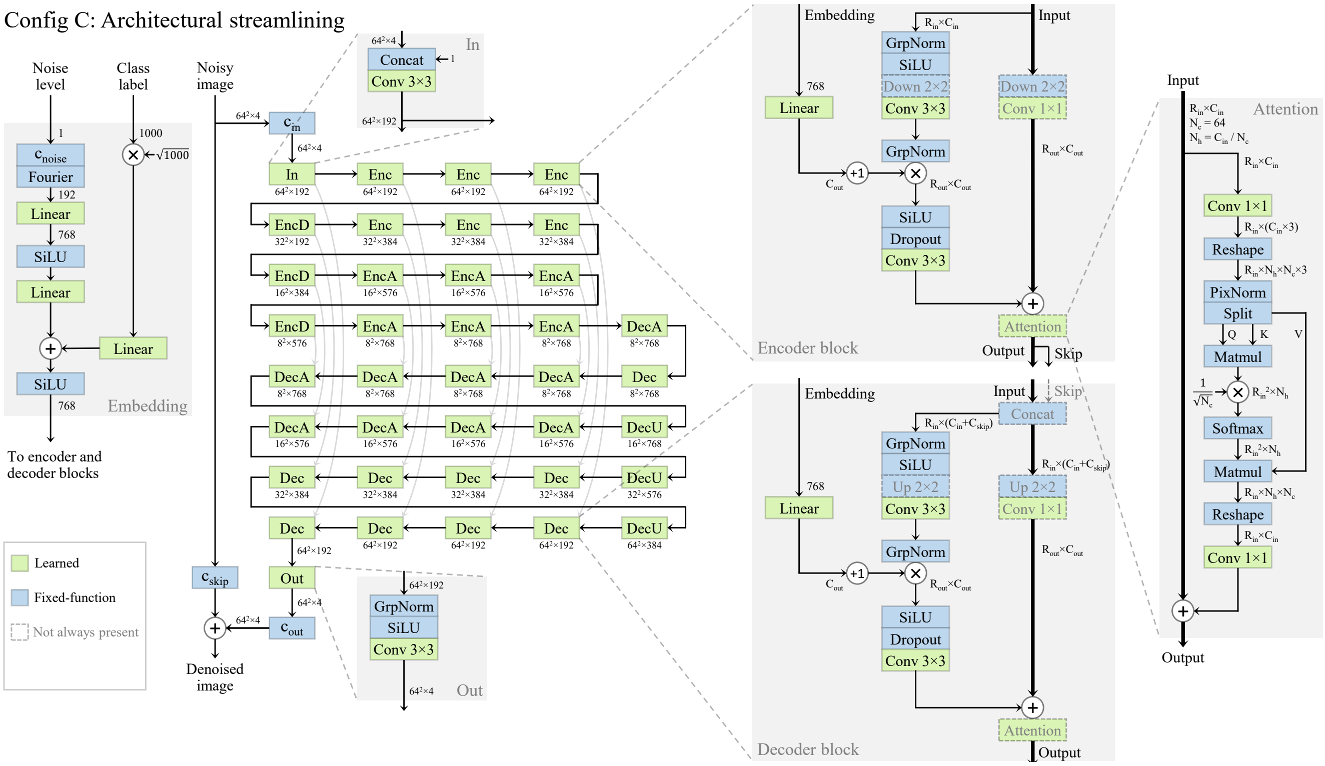
$$= \frac{1}{2} \sum_i \left[\frac{\mathcal{L}_i}{\sigma_i^2} + \ln \sigma_i^2 \right]. \tag{18}$$

Intuitively, the contribution of $\mathcal{L}_i$ is weighted down if the model is uncertain about task i, i.e., if σ_i is high. At the same time, the model is penalized for this uncertainty, encouraging σ_i to be as low as possible.

In practice, it can be quite challenging for typical optimizers — such as Adam — to handle σ_i directly due to the logarithm and the requirement that $\sigma_i > 0$. A more convenient formula [18] is obtained by rewriting Equation 18 in terms of log variance $u_i = \ln \sigma_i^2$:

$$\mathcal{L} = \frac{1}{2} \sum_i \left[\frac{\mathcal{L}_i}{e^{u_i}} + u_i \right] \tag{19}$$

$$\propto \sum_i \left[\frac{\mathcal{L}_i}{e^{u_i}} + u_i \right], \tag{20}$$



Number of GPUs	32	Learning rate max (α_{ref})	0.0002	Adam β_1	0.9	FID	6.96
Minibatch size	2048	Learning rate decay (t_{ref})	∞	Adam β_2	0.99	EMA length (σ_{rel})	0.075
Duration (Mimg)	2147.5	Learning rate rampup (Mimg)	10	Loss scaling	100	Model capacity (Mparams)	277.8
Mixed-precision (FP16)	full	Noise distribution mean (P_{mean})	-0.4	Attention res.	16, 8	Model complexity (Gflops)	100.3
Dropout probability	10%	Noise distribution std. (P_{std})	1.0	Attention blocks	15	Sampling cost (Tflops)	7.58

Figure 18. Full architecture diagram and hyperparameters for CONFIG C (Architectural streamlining).

where we have dropped the constant multiplier $1/2$, as it has no effect on the optimum.

Continuous generalization. For the purpose of applying Equation 20 to the EDM loss in Equation 14, we consider each noise level σ to represent a different task. This means that instead of a discrete number of tasks, we are faced with an infinite continuum of tasks $0 < \sigma < \infty$. In accordance to Equation 14, we consider the loss corresponding to task σ to be $\lambda(\sigma)\mathcal{L}(D_\theta;\sigma)$, leading to the following overall loss:

$$\mathcal{L}(D_\theta, u) \;=\; \mathbb{E}_\sigma\left[\frac{\lambda(\sigma)}{e^{u(\sigma)}}\mathcal{L}(D_\theta;\sigma) + u(\sigma)\right], \qquad (21)$$

where we employ a continuous uncertainty function $u(\sigma)$ instead of a discrete set of scalars $\{u_i\}$.

In practice, we implement $u(\sigma)$ as a simple one-layer MLP (not shown in Figure 17) that is trained alongside the main denoiser network and discarded afterwards. The MLP evaluates $c_{\text{noise}}(\sigma)$ as defined by Equation 8, computes Fourier features for the resulting scalar (see Appendix B.3), and feeds the resulting feature vector through a fully-connected layer that outputs one scalar. All practical details of the MLP, including initialization, magnitude-preserving scaling, and forced weight normalization, follow the choices made in our training configurations (Appendices B.2–B.7).

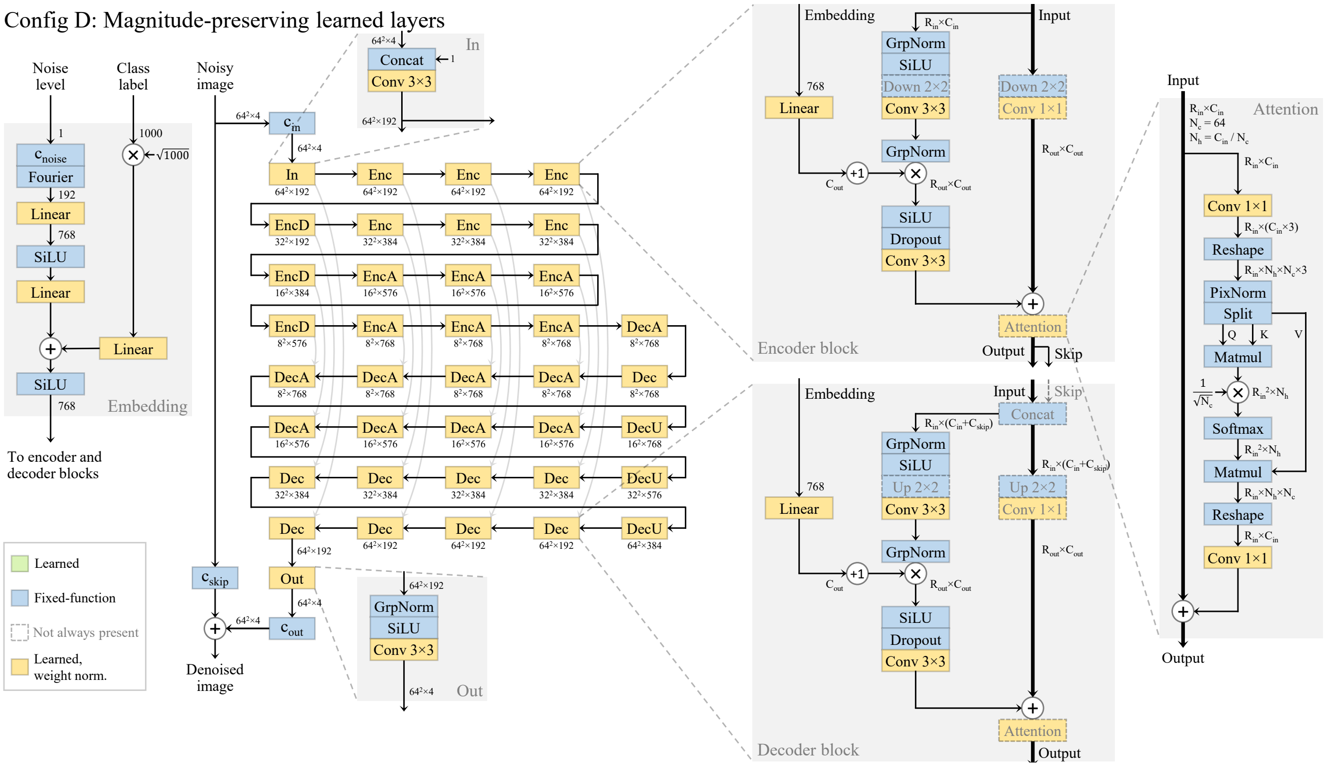
Intuitive interpretation. To gain further insight onto the meaning of Equation 21, let us solve for the minimum of $\mathcal{L}(D_\theta, u)$ by setting its derivative to zero with respect to $u(\sigma)$:

$$\begin{aligned} 0 \;&=\; \frac{\mathrm{d}\mathcal{L}(D_\theta, u)}{\mathrm{d}u(\sigma)} && (22)\\ &=\; \frac{\mathrm{d}}{\mathrm{d}u(\sigma)}\left[\frac{\lambda(\sigma)}{e^{u(\sigma)}}\mathcal{L}(D_\theta;\sigma) + u(\sigma)\right] && (23)\\ &=\; -\frac{\lambda(\sigma)}{e^{u(\sigma)}}\mathcal{L}(D_\theta;\sigma) + 1, && (24)\end{aligned}$$

which leads to

$$\begin{aligned} e^{u(\sigma)} \;&=\; \lambda(\sigma)\mathcal{L}(D_\theta;\sigma) && (25)\\ u(\sigma) \;&=\; \ln\mathcal{L}(D_\theta;\sigma) + \ln\lambda(\sigma). && (26)\end{aligned}$$

In other words, $u(\sigma)$ effectively keeps track of how $\mathcal{L}(D_\theta;\sigma)$ evolves over time. Plugging Equation 25 back into Equa-



Number of GPUs	32	Learning rate max (α_{ref})	0.0100	Adam β_1	0.9	FID	3.75
Minibatch size	2048	Learning rate decay (t_{ref})	∞	Adam β_2	0.99	EMA length (σ_{rel})	0.225
Duration (Mimg)	2147.5	Learning rate rampup (Mimg)	10	Loss scaling	1	Model capacity (Mparams)	277.8
Mixed-precision (FP16)	full	Noise distribution mean (P_{mean})	-0.4	Attention res.	16, 8	Model complexity (Gflops)	101.2
Dropout probability	10%	Noise distribution std. (P_{std})	1.0	Attention blocks	15	Sampling cost (Tflops)	7.64

Figure 19. Full architecture diagram and hyperparameters for CONFIG D (Magnitude-preserving learned layers).

tion 21, we arrive at an alternative interpretation of the overall training loss:

$$\mathcal{L}(D_\theta, u) \;=\; \mathbb{E}_\sigma\left(\frac{\lambda(\sigma)\mathcal{L}(D_\theta;\sigma)}{\big[\lambda(\sigma)\mathcal{L}(D_\theta;\sigma)\big]} + \big[u(\sigma)\big]\right) \qquad (27)$$

$$\;=\; \mathbb{E}_\sigma\frac{\mathcal{L}(D_\theta;\sigma)}{\big[\mathcal{L}(D_\theta;\sigma)\big]} + \big[\mathbb{E}_\sigma u(\sigma)\big], \qquad (28)$$

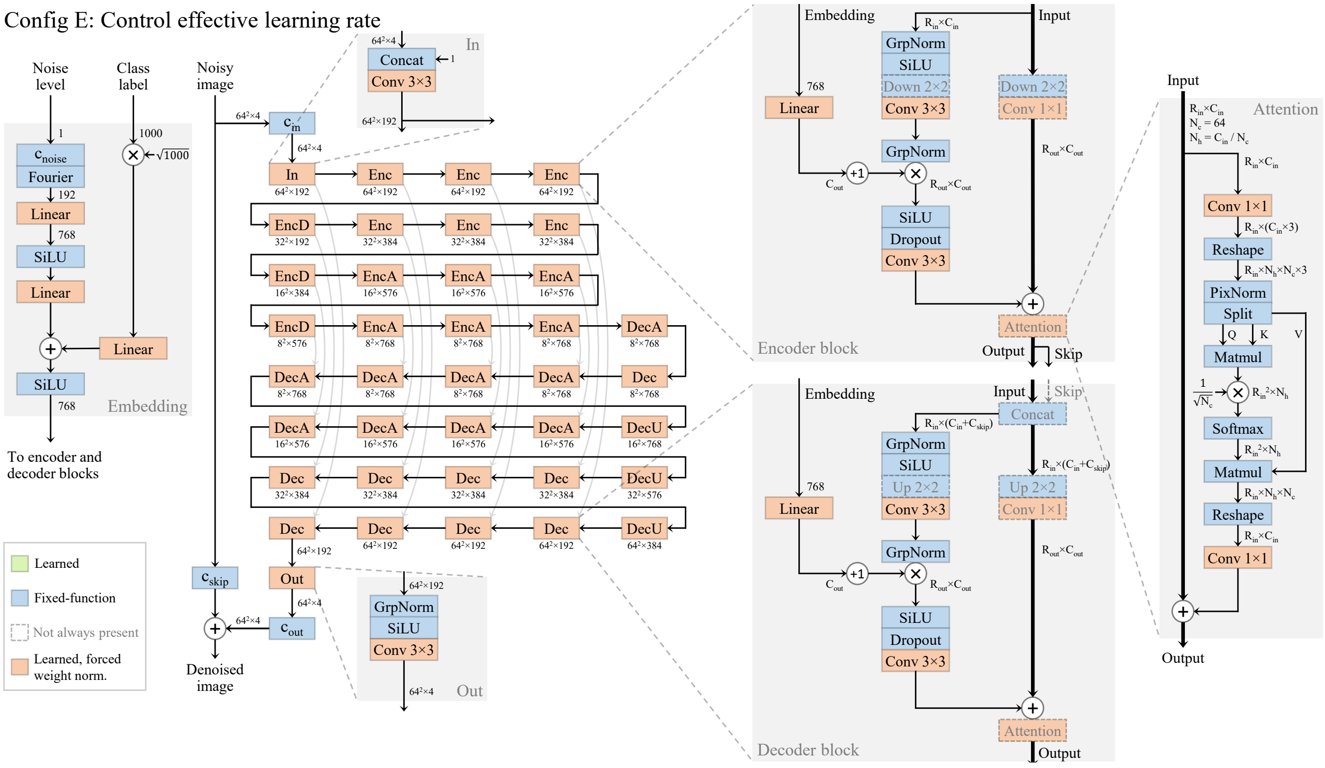
where the bracketed expressions are treated as constants when computing $\nabla_\theta\mathcal{L}(D_\theta, u)$. In other words, Equation 21 effectively scales the gradients originating from noise level σ by the reciprocal of $\mathcal{L}(D_\theta;\sigma)$, equalizing their contribution between noise levels and over time.

Note that the optimum of Equations 21 and 28 with respect to θ does not depend on the choice of $\lambda(\sigma)$. In theory, we could thus drop $\lambda(\sigma)$ altogether, i.e., set $\lambda(\sigma) = 1$. We have tested this in practice and found virtually no impact on the resulting FID or convergence speed. However, we choose to keep $\lambda(\sigma)$ defined according to Equation 15 as a practical safety precaution; Equation 28 only becomes effective once $u(\sigma)$ has converged reasonably close to the optimum, so the choice of $\lambda(\sigma)$ is still relevant at the beginning of the training.

B.3. Architectural streamlining (CONFIG C)

The network architecture of CONFIG B contains several different types of trainable parameters that each behave in a slightly different way: weights and biases of three kinds (uniform-initialized, zero-initialized, and self-attention) as well as group normalization scaling parameters and class embeddings. Our goal in CONFIG C is eliminate these differences and make all the remaining parameters behave more or less identically. To this end, we make several changes to the architecture that can be seen by comparing Figures 17 and 18.

Biases and group normalizations. We have found that we can simply remove all biases with no ill effects. We do this for all convolutions, fully-connected layers, and group normalization layers in the denoiser network as well as in the loss weighting MLP (Equation 21). In theory, this could potentially lead to reduced expressive power of the network, especially when sensing the overall scale of the input values. Even though we have not seen this to be an issue in practice, we mitigate the danger by concatenating an additional channel of constant 1 to the incoming noisy image in the input



Number of GPUs	32	Learning rate max (α_{ref})	0.0100	Adam β_1	0.9	FID	3.02
Minibatch size	2048	Learning rate decay (t_{ref})	70000	Adam β_2	0.99	EMA length (σ_{rel})	0.145
Duration (Mimg)	2147.5	Learning rate rampup (Mimg)	10	Loss scaling	1	Model capacity (Mparams)	277.8
Mixed-precision (FP16)	full	Noise distribution mean (P_{mean})	-0.4	Attention res.	16, 8	Model complexity (Gflops)	101.2
Dropout probability	10%	Noise distribution std. (P_{std})	1.0	Attention blocks	15	Sampling cost (Tflops)	7.64

Figure 20. Full architecture diagram and hyperparameters for CONFIG E (Control effective learning rate).

block (“In”).

Furthermore, we remove all other bias-like constructs for consistency; namely, the dynamic conditioning offset derived from the embedding vector in the encoder and decoder blocks and the subtraction of the empirical mean in group normalization. We further simplify the group normalization layers by removing their learned scale parameter. After these changes, the operation becomes

$$b_{x,y,c,g} = \frac{a_{x,y,c,g}}{\sqrt{\frac{1}{N_x N_y N_c} \sum_{i,j,k} a_{i,j,k,g}^2 + \epsilon}}, \tag{29}$$

where $a_{x,y,c,g}$ and $b_{x,y,c,g}$ denote the incoming and outgoing activations, respectively, for pixel (x, y), channel c, and group g, and N_x, N_y, and N_c indicate their corresponding dimensions. We set $\epsilon = 10^{-4}$.

Cosine attention. The $1{\times}1$ convolutions responsible for producing the query and key vectors for self-attention behave somewhat differently compared to the other convolutions. This is because the resulting values of $w_{i,j}$ (Equation 12) scale quadratically with respect to the overall magnitude of the convolution weights, as opposed to linear scaling in other convolutions. We eliminate this discrepancy by utilizing cosine attention [8, 26, 28]. In practice, we do this by replacing the group normalization, executed right before the convolution, with pixelwise feature vector normalization [14] (“PixelNorm”), executed right after it. This operation is defined as

$$b_{x,y,c} = \frac{a_{x,y,c}}{\sqrt{\frac{1}{N_c} \sum_i a_{x,y,i}^2 + \epsilon}}, \tag{30}$$

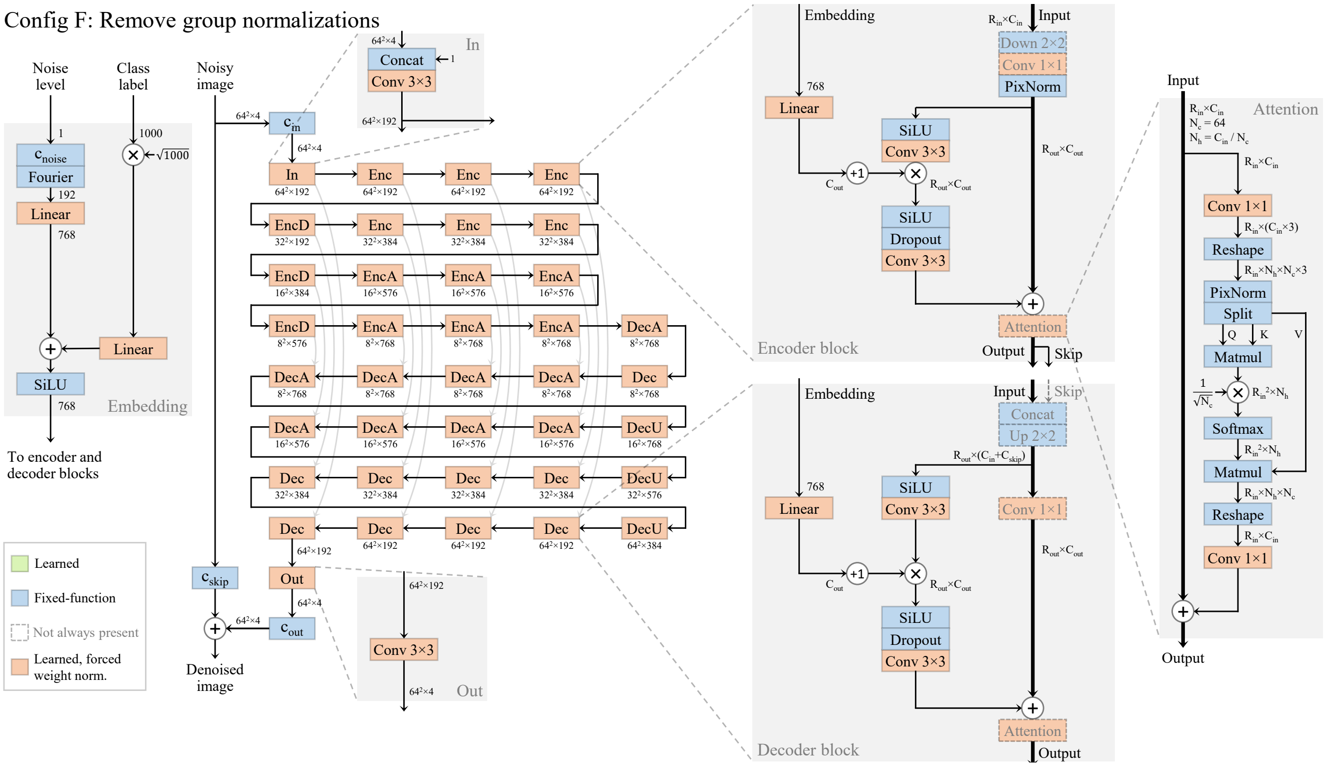
where we use $\epsilon = 10^{-4}$, similar to Equation 29.

To gain further insight regarding the effect of this normalization, we note that, ignoring ϵ, Equation 30 can be equivalently written as

$$\mathbf{b}_{x,y} = \sqrt{N_c}\, \frac{\mathbf{a}_{x,y}}{\|\mathbf{a}_{x,y}\|_2}. \tag{31}$$

Let us denote the normalized query and key vectors by $\hat{\mathbf{q}}_i$ and $\hat{\mathbf{k}}_j$, respectively. Substituting Equation 31 into Equation 12 gives

$$\begin{aligned} w_{i,j} &= \frac{1}{\sqrt{N_c}} \left\langle \hat{\mathbf{q}}_i, \hat{\mathbf{k}}_j \right\rangle && (32) \\ &= \frac{1}{\sqrt{N_c}} \left\langle \sqrt{N_c}\, \frac{\mathbf{q}_i}{\|\mathbf{q}_i\|_2}, \sqrt{N_c}\, \frac{\mathbf{k}_j}{\|\mathbf{k}_j\|_2} \right\rangle && (33) \end{aligned}$$



Number of GPUs	32	Learning rate max (α_{ref})	0.0100	Adam β_1	0.9	FID	2.71
Minibatch size	2048	Learning rate decay (t_{ref})	70000	Adam β_2	0.99	EMA length (σ_{rel})	0.100
Duration (Mimg)	2147.5	Learning rate rampup (Mimg)	10	Loss scaling	1	Model capacity (Mparams)	280.2
Mixed-precision (FP16)	full	Noise distribution mean (P_{mean})	-0.4	Attention res.	16, 8	Model complexity (Gflops)	102.1
Dropout probability	10%	Noise distribution std. (P_{std})	1.0	Attention blocks	15	Sampling cost (Tflops)	7.69

Figure 21. Full architecture diagram and hyperparameters for CONFIG F (Remove group normalizations).

$$= \sqrt{N_c}\,\cos(\phi_{i,j})\,, \tag{34}$$

where $\phi_{i,j}$ denotes the angle between $\mathbf{q}_i$ and $\mathbf{k}_j$. In other words, the attention weights are now determined exclusively by the *directions* of the query and key vectors, and their lengths no longer have any effect. This curbs the uncontrolled growth of $w_{i,j}$ during training and enables using 16-bit floating point throughout the entire self-attention block.

Other changes. To unify the behavior of the remaining trainable parameters, we change the zero-initialized layers (dark green) and the class embeddings to use the same uniform initialization as the rest of the layers. In order to retain the same overall magnitude after the class embedding layer, we scale the incoming one-hot class labels by $\sqrt{N}$ so that the result is of unit variance, i.e., $\frac{1}{N}\sum_i a_i^2 = 1$.

Finally, we replace ADM's original timestep embedding layer with the more standard Fourier features [39]. Concretely, we compute feature vector $\mathbf{b}$ based on the incoming scalar $a = c_{\text{noise}}(\sigma)$ as

$$\mathbf{b} \;=\; \begin{bmatrix} \cos\big(2\pi(f_1\,a + \varphi_1)\big) \\ \cos\big(2\pi(f_2\,a + \varphi_2)\big) \\ \vdots \\ \cos\big(2\pi(f_N a + \varphi_N)\big) \end{bmatrix}, \tag{35}$$

$$\text{where}\;\; f_i \sim \mathcal{N}(0,1) \;\;\text{and}\;\; \varphi_i \sim \mathcal{U}(0,1). \tag{36}$$

After initialization, we treat the frequencies $\{f_i\}$ and phases $\{\varphi_i\}$ as constants.

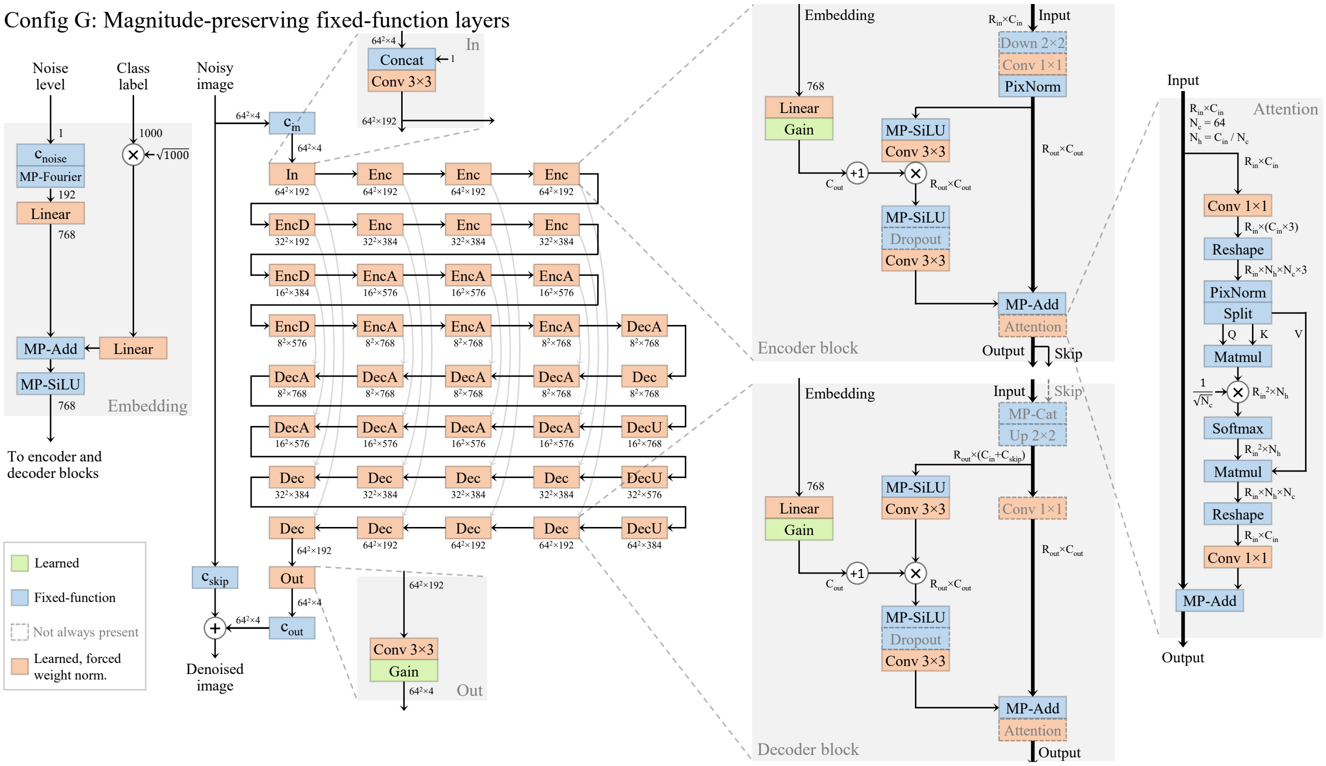
B.4. Magnitude-preserving learned layers (CONFIG D)

In CONFIG D, we modify all learned layers according to our magnitude-preserving design principle as shown in Figure 19. Let us consider a fully-connected layer with input activations $\mathbf{a} = [a_j]^\top$ and output activations $\mathbf{b} = [b_i]^\top$. The operation of the layer is

$$\mathbf{b} \;=\; \mathbf{W}\mathbf{a}\,, \tag{37}$$

where $\mathbf{W} = [\mathbf{w}_i]$ is a trainable weight matrix. We can equivalently write this in terms of a single output element:

$$b_i \;=\; \mathbf{w}_i\,\mathbf{a}\,. \tag{38}$$



Number of GPUs	32	Learning rate max (α_{ref})	0.0100	Adam β_1	0.9	FID	2.56
Minibatch size	2048	Learning rate decay (t_{ref})	70000	Adam β_2	0.99	EMA length (σ_{rel})	0.130
Duration (Mimg)	2147.5	Learning rate rampup (Mimg)	10	Loss scaling	1	Model capacity (Mparams)	280.2
Mixed-precision (FP16)	full	Noise distribution mean (P_{mean})	-0.4	Attention res.	16, 8	Model complexity (Gflops)	102.2
Dropout probability	0%	Noise distribution std. (P_{std})	1.0	Attention blocks	15	Sampling cost (Tflops)	7.70

Figure 22. Full architecture diagram and hyperparameters for CONFIG G (Magnitude-preserving fixed-function layers).

The same definition extends to convolutional layers by applying Equation 38 independently to each output element. In this case, the elements of $\mathbf{a}$ correspond to the activations of each input pixel within the support for the convolution kernel, i.e., $\dim(\mathbf{a}) = N_j = N_c\, k^2$, where N_c is the number of input channels and k is the size of the convolution kernel.

Our goal is to modify Equation 38 so that it preserves the overall magnitude of the input activations, without looking at their actual contents. Let us start by calculating the standard deviation of b_i, assuming that $\{a_i\}$ are mutually uncorrelated and of equal standard deviation σ_a:

$$\sigma_{b_i} = \sqrt{\mathrm{Var}[b_i]} \tag{39}$$

$$= \sqrt{\mathrm{Var}[\mathbf{w}_i\, \mathbf{a}]} \tag{40}$$

$$= \sqrt{\textstyle\sum_j w_{ij}^2\, \mathrm{Var}[a_j]} \tag{41}$$

$$= \sqrt{\textstyle\sum_j w_{ij}^2\, \sigma_a^2} \tag{42}$$

$$= \|\mathbf{w}_i\|_2\, \sigma_a. \tag{43}$$

To make Equation 38 magnitude-preserving, we scale its output so that it has the same standard deviation as the input:

$$\hat{b}_i = \frac{\sigma_a}{\sigma_{b_i}}\, b_i \tag{44}$$

$$= \frac{\sigma_a}{\|\mathbf{w}_i\|_2\, \sigma_a}\, \mathbf{w}_i\, \mathbf{a} \tag{45}$$

$$= \underbrace{\frac{\mathbf{w}_i}{\|\mathbf{w}_i\|_2}}_{=:\, \hat{\mathbf{w}}_i}\, \mathbf{a}. \tag{46}$$

In other words, we simply normalize each $\mathbf{w}_i$ to unit length before use. In practice, we introduce $\epsilon = 10^{-4}$ to avoid numerical issues, similar to Equations 29 and 30:

$$\hat{\mathbf{w}}_i = \frac{\mathbf{w}_i}{\|\mathbf{w}_i\|_2 + \epsilon}. \tag{47}$$

Given that $\hat{b}_i$ is now agnostic to the scale of $\mathbf{w}_i$, we initialize $w_{i,j} \sim \mathcal{N}(0,1)$ so that the weights of all layers are roughly of the same magnitude. This implies that in the early stages of training, when the weights remain close to their initial magnitude, the updates performed by Adam [19] will also have roughly equal impact across the entire model, similar to the concept of *equalized learning rate* [14]. Since the weights are now larger in magnitude, we have to increase the learning rate as well. We therefore set $\alpha_{\text{ref}} = 0.0100$ instead of 0.0002.

Comparison to previous work. Our approach is closely related to *weight normalization* [36] and *weight standardization* [31]. Reusing the notation from Equation 46, Salimans and Kingma [36] define weight normalization as

$$\hat{\mathbf{w}}_i = \frac{g_i}{\|\mathbf{w}_i\|_2} \mathbf{w}_i, \tag{48}$$

where g_i is a learned per-channel scaling factor that is initialized to one. The original motivation of Equation 48 is to reparameterize the weight tensor in order to speed up convergence, without affecting its expressive power. As such, the value of g_i is free to drift over the course of training, potentially leading to imbalances in the overall activation magnitudes. Our motivation, on the other hand, is to explicitly avoid such imbalances by removing any direct means for the optimization to change the magnitude of $\hat{b}_i$.

Qiao et al. [31], on the other hand, define weight standardization as

$$\hat{\mathbf{w}}_i = \frac{\mathbf{w}_i - \mu_i}{\sigma_i} \text{, where} \tag{49}$$

$$\mu_i = \frac{1}{N}\sum\nolimits_j w_{i,j} \tag{50}$$

$$\sigma_i = \sqrt{\frac{1}{N}\sum\nolimits_j w_{i,j}^2 - \mu_i^2 + \epsilon}\,, \tag{51}$$

intended to serve as a replacement for batch normalization in the context of micro-batch training. In practice, we suspect that Equation 49 would probably work just as well as Equation 46 for our purposes. However, we prefer to keep the formula as simple as possible with no unnecessary moving parts.

Effect on the gradients. One particularly useful property of Equation 46 is that it projects the gradient of $\mathbf{w}_i$ to be perpedicular to $\mathbf{w}_i$ itself. Let us derive a formula for the gradient of loss $\mathcal{L}$ with respect to $\mathbf{w}_i$:

$$\nabla_{\mathbf{w}_i}\mathcal{L} = \nabla_{\mathbf{w}_i}\hat{\mathbf{w}}_i \cdot \nabla_{\hat{\mathbf{w}}_i}\mathcal{L} \tag{52}$$

$$= \nabla_{\mathbf{w}_i}\left[\frac{\mathbf{w}_i}{\|\mathbf{w}_i\|_2}\right]\nabla_{\hat{\mathbf{w}}_i}\mathcal{L}. \tag{53}$$

We will proceed using the quotient rule

$$\left[\frac{f}{g}\right]' = \frac{f'g - fg'}{g^2}, \tag{54}$$

where

$$f = \mathbf{w}_i, \qquad f' = \nabla_{\mathbf{w}_i}\mathbf{w}_i \qquad = \mathbf{I} \tag{55}$$

$$g = \|\mathbf{w}_i\|_2\,, \;\; g' = \nabla_{\mathbf{w}_i}\|\mathbf{w}_i\|_2 = \frac{\mathbf{w}_i^\top}{\|\mathbf{w}_i\|_2}. \tag{56}$$

Plugging this back into Equation 53 gives us

$$\nabla_{\mathbf{w}_i}\mathcal{L} = \left[\frac{f'g - fg'}{g^2}\right]\nabla_{\hat{\mathbf{w}}_i}\mathcal{L} \tag{57}$$

$$= \left[\frac{\mathbf{I}\,\|\mathbf{w}_i\|_2 - \mathbf{w}_i\frac{\mathbf{w}_i^\top}{\|\mathbf{w}_i\|_2}}{\|\mathbf{w}_i\|_2^2}\right]\nabla_{\hat{\mathbf{w}}_i}\mathcal{L} \tag{58}$$

$$= \frac{1}{\|\mathbf{w}_i\|_2}\left[\mathbf{I} - \frac{\mathbf{w}_i\mathbf{w}_i^\top}{\|\mathbf{w}_i\|_2^2}\right]\nabla_{\hat{\mathbf{w}}_i}\mathcal{L}. \tag{59}$$

The bracketed expression in Equation 59 corresponds to a projection matrix that keeps the incoming vector otherwise unchanged, but forces it to be perpendicular to $\mathbf{w}_i$, i.e., $\langle\mathbf{w}_i, \nabla_{\mathbf{w}_i}\mathcal{L}\rangle = 0$. In other words, gradient descent optimization will not attempt to modify the length of $\mathbf{w}_i$ directly. However, the length of $\mathbf{w}_i$ can still change due to discretization errors resulting from finite step size.

B.5. Controlling effective learning rate (CONFIG E)

In CONFIG D, we have effectively constrained all weight vectors of our model to lie on the unit hypersphere, i.e., $\|\hat{\mathbf{w}}_i\|_2 = 1$, as far as evaluating $D_\theta(\boldsymbol{x};\sigma)$ is concerned. However, the magnitudes of the raw weight vectors, i.e., $\|\mathbf{w}_i\|_2$, are still relevant during training due to their effect on $\nabla_{\mathbf{w}_i}\mathcal{L}$ (Equation 59). Even though we have initialized $\mathbf{w}_i$ so that these magnitudes are initially balanced across the layers, there is nothing to prevent them from drifting away from this ideal over the course of training. This is problematic since the relative impact of optimizer updates, i.e., the *effective learning rate*, can vary uncontrollably across the layers and over time. In CONFIG E, we eliminate this drift through *forced weight normalization* as shown in Figure 20, and gain explicit control over the effective learning rate.

Growth of weight magnitudes. As noted by Salimans and Kingma [36], Equations 46 and 59 have the side effect that they cause the norm of $\mathbf{w}_i$ to increase monotonically after each training step. As an example, let us consider standard gradient descent with learning rate α. The update rule is defined as

$$\mathbf{w}_i' = \mathbf{w}_i - \alpha\nabla_{\mathbf{w}_i}\mathcal{L} \tag{60}$$

$$\mathbf{w}_i \leftarrow \mathbf{w}_i'. \tag{61}$$

We can use the Pythagorean theorem to calculate the norm of the updated weight vector $\mathbf{w}_i'$:

$$\|\mathbf{w}_i'\|_2^2 = \|\mathbf{w}_i - \alpha\nabla_{\mathbf{w}_i}\mathcal{L}\|_2^2 \tag{62}$$

$$= \|\mathbf{w}_i\|_2^2 + \alpha^2\|\nabla_{\mathbf{w}_i}\mathcal{L}\|_2^2 - 2\alpha\underbrace{\langle\mathbf{w}_i, \nabla_{\mathbf{w}_i}\mathcal{L}\rangle}_{=0} \tag{63}$$

Algorithm 1 PyTorch code for forced weight normalization.

```
def normalize(x, eps=1e-4):
    dim = list(range(1, x.ndim))
    n = torch.linalg.vector_norm(x, dim=dim, keepdim=True)
    alpha = np.sqrt(n.numel() / x.numel())
    return x / torch.add(eps, n, alpha=alpha)

class Conv2d(torch.nn.Module):
    def __init__(self, C_in, C_out, k):
        super().__init__()
        w = torch.randn(C_out, C_in, k, k)
        self.weight = torch.nn.Parameter(w)

    def forward(self, x):
        if self.training:
            with torch.no_grad():
                self.weight.copy_(normalize(self.weight))
        fan_in = self.weight[0].numel()
        w = normalize(self.weight) / np.sqrt(fan_in)
        x = torch.nn.functional.conv2d(x, w, padding='same')
        return x
```

$$= \left\|\mathbf{w}_i\right\|_2^2 + \alpha^2 \left\|\nabla_{\mathbf{w}_i}\mathcal{L}\right\|_2^2 \tag{64}$$

$$\geq \left\|\mathbf{w}_i\right\|_2^2. \tag{65}$$

In other words, the norm of $\mathbf{w}_i$ will necessarily increase at each step unless $\nabla_{\mathbf{w}_i}\mathcal{L} = \mathbf{0}$. A similar phenomenon has been observed with optimizers like Adam [19], whose updates do not maintain strict orthogonality, as well as in numerous scenarios that do not obey Equation 46 exactly. The effect is apparent in our CONFIG C (Figure 3) as well.

Forced weight normalization. Given that the normalization and initialization discussed in Appendix B.4 are already geared towards constraining the weight vectors to a hypersphere, we take this idea to its logical conclusion and perform the entire optimization strictly under such a constraint.

Concretely, we require $\|\mathbf{w}_i\|_2 = \sqrt{N_j}$ to be true for each layer after each training step, where N_j is the dimension of $\mathbf{w}_i$, i.e., the fan-in. Equation 59 already constrains $\nabla_{\mathbf{w}_i}\mathcal{L}$ to lie on the tangent plane with respect to this constraint; the only missing piece is to guarantee that the constraint itself is satisfied by Equation 61. To do this, we modify the formula to forcefully re-normalize $\mathbf{w}'_i$ before assigning it back to $\mathbf{w}_i$:

$$\mathbf{w}_i \;\leftarrow\; \sqrt{N_j}\,\frac{\mathbf{w}'_i}{\|\mathbf{w}'_i\|_2}. \tag{66}$$

Note that Equation 66 is agnostic to the exact definition of $\mathbf{w}'_i$, so it is readily compatible with most of the commonly used optimizers. In theory, it makes no difference whether the normalization is done before or after the actual training step. In practice, however, the former leads to a very simple and concise PyTorch implementation, shown in Algorithm 1.

Learning rate decay. Let us step back and consider CONFIG D again for a moment, focusing on the overall effect that $\|\mathbf{w}_i\|_2$ had on the training dynamics. Networks where magnitudes of weights have no effect on activations have previously been studied by, e.g., van Laarhoven [22]. In these networks, the only meaningful progress is made in the *angular* direction of weight vectors. This has two consequences for training dynamics: First, the gradients seen by the optimizer are inversely proportional to the weight magnitude. Second, the loss changes slower at larger magnitudes, as more distance needs to be covered for the same angular change. Effectively, both of these phenomena can be interpreted as downscaling the effective learning rate as a function of the weight magnitude. Adam [19] counteracts the first effect by approximately normalizing the gradient magnitudes, but it does not address the second.

From this perspective, we can consider CONFIG D to have effectively employed an implicit learning rate decay: The larger the weights have grown (Figure 3), the smaller the effective learning rate. In general, learning rate decay is considered desirable in the sense that it enables the training to converge closer and closer to the optimum despite the stochastic nature of the gradients [19, 36]. However, we argue that the implicit form of learning rate decay imposed by Equation 65 is not ideal, because it can lead to uncontrollable and unequal drift between layers.

With forced weight normalization in CONFIG E and onwards, the drift is eliminated and the effective learning rate is directly proportional to the specified learning rate α. Thus, in order to have the learning rate decay, we have to explicitly modify the value of α over time. We choose to use the commonly advocated inverse square root decay schedule [19]:

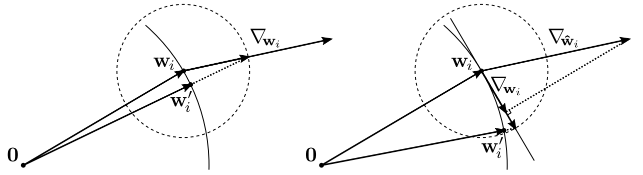
$$\alpha(t) \;=\; \frac{\alpha_{\text{ref}}}{\sqrt{\max(t/t_{\text{ref}}, 1)}}, \tag{67}$$

where the learning rate initially stays at α_{ref} and then starts decaying after t_{ref} training iterations. The constant learning rate schedule of CONFIGS A–D can be seen as a special case of Equation 67 with $t_{\text{ref}} = \infty$.

In the context of Table 1, we use $\alpha_{\text{ref}} = 0.0100$ and $t_{\text{ref}} = 70000$. We have, however, found that the optimal choices depend heavily on the capacity of the network as well as the dataset (see Table 6).

Discussion. It is worth noticing that we normalize the weight vectors *twice* during each training step: first to obtain $\hat{\mathbf{w}}_i$ in Equation 46 and then to constrain $\mathbf{w}'_i$ in Equation 66. This is also reflected by the two calls to `normalize()` in Algorithm 1.

The reason why Equation 46 is still necessary despite Equation 66 is that it ensures that Adam's variance estimates are computed for the actual tangent plane steps. In other words, Equation 46 lets Adam "know" that it is supposed to operate under the fixed-magnitude constraint. If we used Equation 66 alone, without Equation 46, the variance



(a) Forced WN only

(b) Forced + standard WN

Figure 23. Illustration of the importance of performing "standard" weight normalization in addition to forcing the weights to a predefined norm. The dashed circle illustrates Adam's target variance for updates — the proportions are greatly exaggerated and the effects of momentum are ignored. **(a)** Forced weight normalization without the standard weight normalization. The raw weight vector $\mathbf{w}_i$ is updated by adding the gradient $\nabla_{\mathbf{w}_i}$ after being scaled by Adam, after which the result is normalized back to the hypersphere (solid arc) yielding new weight vector $\mathbf{w}_i'$. Adam's variance estimate includes the non-tangent component of the gradient, and the resulting weight update is significantly smaller than intended. **(b)** With standard weight normalization, the gradient $\nabla_{\mathbf{w}_i}$ is obtained by projecting the raw gradient $\nabla_{\hat{\mathbf{w}}_i}$ onto the tangent plane perpendicular to $\mathbf{w}_i$. Adam's variance estimate now considers this projected gradient, resulting in the correct step size; the effect of normalization after update is close to negligible from a single step's perspective.

estimates would be corrupted by the to-be erased normal component of the raw gradient vectors, leading to considerably smaller updates of an uncontrolled magnitude. See Figure 23 for an illustration.

Furthermore, we intentionally force the raw weights $\mathbf{w}_i$ to have the norm $\sqrt{N_j}$, while weight normalization further scales them to norm 1. The reason for this subtle but important difference is, again, compatibility with the Adam optimizer. Adam approximately normalizes the gradient updates so that they are proportional to $\sqrt{N_j}$. We normalize the weights to the same scale, so that the *relative* magnitude of the update becomes independent of N_j. This eliminates an implicit dependence between the learning rate and the layer size. Optimizers like LARS [43] and Fromage [2] build on a similar motivation, and explicitly scale the norm of the gradient updates to a fixed fraction of the weight norm.

Finally, Equation 46 is also quite convenient due to its positive interaction with EMA. Even though the raw values of $\mathbf{w}_i$ are normalized at each training step by Equation 66, their weighted averages are not. To correctly account for our fixed-magnitude constraint, the averaging must also happen along the surface of the corresponding hypersphere. However, we do not actually need to change the averaging itself in any way, because this is already taken care of by Equation 46: Even if the magnitudes of the weight vectors change considerably as a result of averaging, they are automatically re-normalized upon use.

Previous work. Several previous works have analyzed the consequences of weight magnitude growth under different settings and proposed various remedies. Weight decay has often been identified as a solution for keeping the magnitudes in check, and its interplay with different normalization schemes and optimizers has been studied extensively [13, 20, 22–24, 32, 42, 45]. Cho and Lee [6] and van Laarhoven [22] consider more direct approaches where the weights are directly constrained to remain in the unit norm hypersphere, eliminating the growth altogether. Arpit et al. [1] also normalize the weights directly, motivated by a desire to reduce the parameter space. Various optimizers [2, 3, 25, 43, 44] also aim for similar effects through weight-relative scaling of the gradient updates.

As highlighted by the above discussion, the success of these approaches can depend heavily on various small but important nuances that may not be immediately evident. As such, we leave a detailed comparison of these approaches as future work.

B.6. Removing group normalizations (CONFIG F)

In CONFIG F, our goal is to remove the group normalization layers that may negatively impact the results due to the fact that they operate across the entire image. We also make a few minor simplifications to the architecture. These changes can be seen by comparing Figures 20 and 21.

Dangers of global normalization. As has been previously noted [15, 16], global normalization that operates across the entire image should be used cautiously. It is firmly at odds with the desire for the model to behave consistently across geometric transformations [16, 40] or when synthesizing objects in different contexts. Such consistency is easiest to achieve if the internal representations of the image contents are capable of being as localized as they need to be, but global normalization entangles the representations of every part of the image by eliminating the first-order statistics across the image. Notably, while attention *allows* the representations to communicate with each other in a way that best fits the task, global normalization *forces* communication to occur, with no way for individual features to avoid it.

This phenomenon has been linked to concrete image artifacts in the context of GANs. Karras et al. [15] found that the AdaIN operation used in StyleGAN was destroying vital information, namely the relative scales of different feature maps, which the model counteracted by creating strong localized spikes in the activations. These spikes manifested as artifacts, and were successfully eliminated by removing global normalization operations. In a different context, Brock et al. [5] show that normalization is not necessary for obtaining high-quality results in image classification. We see no reason why it should be necessary or even beneficial in diffusion models, either.

Our approach. Having removed the drift in activation magnitudes, we find that we can simply remove all group normalization layers with no obvious downsides. In particular, doing this for the decoder improves the FID considerably, which we suspect to be related to the fact that the absolute scale of the individual output pixels is quite important for the training loss (Equation 13). The network has to start preparing the correct scales towards the end of the U-Net, and explicit normalization is likely to make this more challenging.

Even though explicit normalization is no longer strictly necessary, we have found that we can further improve the results slightly through pixelwise feature vector normalization (Equation 30). Our hypothesis is that a small amount of normalization helps by counteracting correlations that would otherwise violate the independence assumption behind Equation 43. We find that the best results are obtained by normalizing the incoming activations at the beginning of each encoder block. This guarantees that the magnitudes on the main path remain standardized despite the series of cumulative adjustments made by the residual and self-attention blocks. Furthermore, this also appears to help in terms of standardizing the magnitudes of the decoder — presumably due to the presence of the U-Net skip connections.

Architectural simplifications. In addition to reworking the normalizations, we make four minor simplifications to other parts of the architecture:

1. Unify the upsampling and downsampling operations of the encoder and decoder blocks by placing them onto the main path.
2. Slightly increase the expressive power of the encoder blocks by moving the 1×1 convolution to the beginning of the main path.
3. Remove the SiLU activation in the final output block.
4. Remove the second fully-connected layer in the embedding network.

These changes are more or less neutral in terms of the FID, but we find it valuable to keep the network as simple as possible considering future work.

B.7. Magnitude-preserving fixed-function layers (CONFIG G)

In CONFIG G, we complete the effort that we started in CONFIG D by extending our magnitude-preserving design principle to cover the remaining fixed-function layers in addition to the learned ones. The exact set of changes can be seen by comparing Figures 21 and 22.

We will build upon the concept of *expected magnitude* that we define by generalizing Equation 3 for multivariate random variable $\mathbf{a}$:

$$\mathcal{M}[\mathbf{a}] \;=\; \sqrt{\frac{1}{N_a}\sum_{i=1}^{N_a}\mathbb{E}\big[a_i^2\big]}. \tag{68}$$

If the elements of $\mathbf{a}$ have zero mean and equal variance, we have $\mathcal{M}[\mathbf{a}]^2 = \mathrm{Var}[a_i]$. If $\mathbf{a}$ is non-random, Equation 68 simplifies to $\mathcal{M}[\mathbf{a}] = \|\mathbf{a}\|_2 / \sqrt{N_a}$. We say that $\mathbf{a}$ is *standardized* iff $\mathcal{M}[\mathbf{a}] = 1$.

Concretely, we aim to achieve two things: First, every input to the network should be standardized, and second, every operation in the network should be such that if its input is standardized, the output is standardized as well. If these two requirements are met, it follows that all activations throughout the entire network are standardized.

Similar to Appendix B.4, we wish to avoid having to look at the actual values of activations, which necessitates making certain simplifying statistical assumptions about them. Even though these assumptions are not strictly true in practice, we find that the end result is surprisingly close to our ideal, as can be seen in the "Activations (mean)" plot for CONFIG G in Figure 15.

Fourier features. Considering the inputs to our network, the noisy image and the class label are already standardized by virtue of having been scaled by $c_{\text{in}}(\sigma)$ (Equation 7) and $\sqrt{N}$ (Appendix B.3), respectively. The Fourier features (Appendix B.3), however, are not. Let us compute the expected magnitude of $\mathbf{b}$ (Equation 35) with respect to the frequencies and phases (Equation 36):

$$\begin{aligned}
\mathcal{M}[\mathbf{b}]^2 &= \frac{1}{N_b}\sum_{i=1}^{N_b}\mathbb{E}\Big[\Big(\cos\big(2\pi(f_i a + \varphi_i)\big)\Big)^2\Big] && (69)\\
&= \mathbb{E}\Big[\Big(\cos\big(2\pi(f_1 a + \varphi_1)\big)\Big)^2\Big] && (70)\\
&= \mathbb{E}\Big[\big(\cos(2\pi\varphi_1)\big)^2\Big] && (71)\\
&= \mathbb{E}\Big[\tfrac{1}{2}\big(1 + \cos(4\pi\varphi_1)\big)\Big] && (72)\\
&= \tfrac{1}{2} + \tfrac{1}{2}\underbrace{\mathbb{E}\big[\cos(4\pi\varphi_1)\big]}_{=\,0} && (73)\\
&= \tfrac{1}{2}. && (74)
\end{aligned}$$

To standardize the output, we thus scale Equation 35 by $1/\mathcal{M}[\mathbf{b}] = \sqrt{2}$:

$$\text{MP-Fourier}(a) \;=\; \begin{bmatrix} \sqrt{2}\cos\big(2\pi(f_1\, a + \varphi_1)\big) \\ \sqrt{2}\cos\big(2\pi(f_2\, a + \varphi_2)\big) \\ \vdots \\ \sqrt{2}\cos\big(2\pi(f_N\, a + \varphi_N)\big) \end{bmatrix}. \tag{75}$$

SiLU. Similar reasoning applies to the SiLU nonlinearity (Equation 9) as well, used throughout the network. Assuming that $\mathbf{a} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$:

$$\begin{aligned}
\mathcal{M}\big[\mathrm{silu}(\mathbf{a})\big]^2 &= \frac{1}{N_a}\sum_{i=1}^{N_a}\mathbb{E}\Big[\big(\mathrm{silu}(a_i)\big)^2\Big] && (76)\\
&= \mathbb{E}\left[\left(\frac{a_1}{1+e^{-a_1}}\right)^2\right] && (77)\\
&= \int_{-\infty}^{\infty}\frac{\mathcal{N}(x;0,1)\,x^2}{(1+e^{-x})^2}\,\mathrm{d}x && (78)\\
&\approx 0.3558 && (79)\\
\mathcal{M}\big[\mathrm{silu}(\mathbf{a})\big] &\approx \sqrt{0.3558} \approx 0.596. && (80)
\end{aligned}$$

Dividing the output accordingly, we obtain

$$\text{MP-SiLU}(\mathbf{a}) = \frac{\mathrm{silu}(\mathbf{a})}{0.596} = \left[\frac{a_i}{0.596\cdot(1+e^{-a_i})}\right]. \quad (81)$$

Sum. Let us consider the weighted sum of two random vectors, i.e., $\mathbf{c} = w_a\mathbf{a} + w_b\mathbf{b}$. We assume that the elements within each vector have equal expected magnitude and that $\mathbb{E}[a_i b_i] = 0$ for every i. Now,

$$\begin{aligned}
\mathcal{M}[\mathbf{c}]^2 &= \frac{1}{N_c}\sum_{i=1}^{N_c}\mathbb{E}\big[(w_a a_i + w_b b_i)^2\big] && (82)\\
&= \frac{1}{N_c}\sum_{i=1}^{N_c}\mathbb{E}\big[w_a^2 a_i^2 + w_b^2 b_i^2 + 2 w_a w_b a_i b_i\big] && (83)\\
&= \frac{1}{N_c}\sum_{i=1}^{N_c}\Big[w_a^2\underbrace{\mathbb{E}\big[a_i^2\big]}_{=\mathcal{M}[\mathbf{a}]^2} + w_b^2\underbrace{\mathbb{E}\big[b_i^2\big]}_{=\mathcal{M}[\mathbf{b}]^2} + 2w_a w_b\underbrace{\mathbb{E}\big[a_i b_i\big]}_{=0}\Big] && (84)\\
&= \frac{1}{N_c}\sum_{i=1}^{N_c}\big[w_a^2\mathcal{M}[\mathbf{a}]^2 + w_b^2\mathcal{M}[\mathbf{b}]^2\big] && (85)\\
&= w_a^2\mathcal{M}[\mathbf{a}]^2 + w_b^2\mathcal{M}[\mathbf{b}]^2. && (86)
\end{aligned}$$

If the inputs are standardized, Equation 86 further simplifies to $\mathcal{M}[\mathbf{c}] = \sqrt{w_a^2 + w_b^2}$. A standardized version of $\mathbf{c}$ is then given by

$$\hat{\mathbf{c}} = \frac{\mathbf{c}}{\mathcal{M}[\mathbf{c}]} = \frac{w_a\mathbf{a} + w_b\mathbf{b}}{\sqrt{w_a^2 + w_b^2}}. \quad (87)$$

Note that Equation 87 is agnostic to the scale of w_a and w_b. Thus, we can conveniently define them in terms of blend factor $t \in [0, 1]$ that can be adjusted on a case-by-case basis. Setting $w_a = (1-t)$ and $w_b = t$, we arrive at our final definition:

$$\text{MP-Sum}(\mathbf{a}, \mathbf{b}, t) = \frac{(1-t)\,\mathbf{a} + t\,\mathbf{b}}{\sqrt{(1-t)^2 + t^2}}. \quad (88)$$

We have found that the best results are obtained by setting $t = 0.3$ in the encoder, decoder, and self-attention blocks, so that the residual path contributes 30% to the result while the main path contributes 70%. In the embedding network $t = 0.5$ seems to work well, leading to equal contribution between the noise level and the class label.

Concatenation. Next, let us consider the concatenation of two random vectors $\mathbf{a}$ and $\mathbf{b}$, scaled by constants w_a and w_b, respectively. The result is given by $\mathbf{c} = w_a\mathbf{a} \oplus w_b\mathbf{b}$, which implies that

$$\begin{aligned}
\mathcal{M}[\mathbf{c}]^2 &= \frac{\sum_{i=1}^{N_c}\mathbb{E}\big[c_i^2\big]}{N_c} && (89)\\
&= \frac{\sum_{i=1}^{N_a}\mathbb{E}\big[w_a^2 a_i^2\big] + \sum_{i=1}^{N_b}\mathbb{E}\big[w_b^2 b_i^2\big]}{N_a + N_b} && (90)\\
&= \frac{w_a^2 N_a \mathcal{M}[\mathbf{a}]^2 + w_b^2 N_b \mathcal{M}[\mathbf{b}]^2}{N_a + N_b}. && (91)
\end{aligned}$$

Note that the contribution of $\mathbf{a}$ and $\mathbf{b}$ in Equation 91 is proportional to N_a and N_b, respectively. If $N_a \gg N_b$, for example, the result will be dominated by $\mathbf{a}$ while the contribution of $\mathbf{b}$ is largely ignored. In our architecture (Figure 22), this situation can arise at the beginning of the decoder blocks when the U-Net skip connection is concatenated into the main path. We argue that the balance between the two branches should be treated as an independent hyperparameter, as opposed to being tied to their respective channel counts.

We first consider the case where we require the two inputs to contribute equally, i.e.,

$$w_a^2 N_a \mathcal{M}[\mathbf{a}]^2 = w_b^2 N_b \mathcal{M}[\mathbf{b}]^2 = C^2, \quad (92)$$

where C is an arbitrary constant. Solving for w_a and w_b:

$$\begin{aligned}
w_a &= \frac{C}{\mathcal{M}[\mathbf{a}]}\cdot\frac{1}{\sqrt{N_a}} && (93)\\
w_b &= \frac{C}{\mathcal{M}[\mathbf{b}]}\cdot\frac{1}{\sqrt{N_b}} && (94)
\end{aligned}$$

Next, we introduce blend factor $t \in [0, 1]$ to allow adjusting the balance between $\mathbf{a}$ and $\mathbf{b}$ on a case-by-case basis, similar to Equation 88:

$$\begin{aligned}
\hat{w}_a &= w_a\,(1-t) = \frac{C}{\mathcal{M}[\mathbf{a}]}\cdot\frac{1-t}{\sqrt{N_a}} && (95)\\
\hat{w}_b &= w_b\,t = \frac{C}{\mathcal{M}[\mathbf{b}]}\cdot\frac{t}{\sqrt{N_b}}. && (96)
\end{aligned}$$

If the inputs are standardized, i.e., $\mathcal{M}[\mathbf{a}] = \mathcal{M}[\mathbf{b}] = 1$, we can solve for the value of C that leads to the output being standardized as well:

$$1 = \mathcal{M}[\mathbf{c}]^2 \quad (97)$$

$$= \frac{\hat{w}_a^2 N_a \mathcal{M}[\mathbf{a}]^2 + \hat{w}_b^2 N_b \mathcal{M}[\mathbf{b}]^2}{N_a + N_b} \tag{98}$$

$$= \frac{\hat{w}_a^2 N_a + \hat{w}_b^2 N_b}{N_a + N_b} \tag{99}$$

$$= \frac{\left[C^2 \frac{(1-t)^2}{N_a}\right] N_a + \left[C^2 \frac{t^2}{N_b}\right] N_b}{N_a + N_b} \tag{100}$$

$$= C^2 \, \frac{(1-t)^2 + t^2}{N_a + N_b}, \tag{101}$$

which yields

$$C \;=\; \sqrt{\frac{N_a + N_b}{(1-t)^2 + t^2}}. \tag{102}$$

Combining Equation 102 with Equations 95 and 96, we arrive at our final definition:

$$\text{MP-Cat}(\mathbf{a}, \mathbf{b}, t) = \sqrt{\frac{N_a + N_b}{(1-t)^2 + t^2}} \cdot \left[\frac{1-t}{\sqrt{N_a}} \, \mathbf{a} \oplus \frac{t}{\sqrt{N_b}} \, \mathbf{b} \right]. \tag{103}$$

In practice, we have found that the behavior of the model is quite sensitive to the choice of t and that the best results are obtained using $t = 0.5$. We hope that the flexibility offered by Equation 103 may prove useful in the future, especially in terms of exploring alternative network architectures.

Learned gain. While our goal of standardizing activations throughout the network is beneficial for the training dynamics, it can also be harmful in cases where it is *necessary* to have $\mathcal{M}[\mathbf{a}] \neq 1$ in order to satisfy the training loss.

We identify two such instances in our network: the raw pixels (F_θ) produced by the final output block ("Out"), and the learned per-channel scaling in the encoder and decoder blocks. In order to allow $\mathcal{M}[\mathbf{a}]$ to deviate from 1, we introduce a simple learned scaling layer at these points:

$$\text{Gain}(\mathbf{a}) \;=\; g\, \mathbf{a}, \tag{104}$$

where g is a learned scalar that is initialized to 0. We have not found it necessary to introduce multiple scaling factors on a per-channel, per-noise-level, or per-class basis. Note that $g = 0$ implies $F_\theta(\boldsymbol{x}; \sigma) = \mathbf{0}$, meaning that $D_\theta(\boldsymbol{x}; \sigma) = \boldsymbol{x}$ at initialization, similar to CONFIGS A–B (see Appendix B.1).

C. Post-hoc EMA details

As discussed in Section 3, our goal is to be able to select the EMA length, or more generally, the model averaging profile, after a training run has completed. This is achieved by storing a number of pre-averaged models during training, after which these pre-averaged models can be linearly combined to obtain a model whose averaging profile has the desired shape and length.

As a related contribution, we present the power function EMA profile that automatically scales according to training time and has zero contribution at $t = 0$.

In this section, we first derive the formulae related to the traditional exponential EMA from first principles, after which we do the same for the power function EMA. We then discuss how to determine the appropriate linear combination of pre-averaged models stored in training snapshots in order to match a given averaging profile, and specifically, to match the power function EMA with a given length.

C.1. Definitions

Let us denote the weights of the network as a function of training time by $\theta(t)$, so that $\theta(0)$ corresponds to the initial state and $\theta(t_c)$ corresponds to the most recent state. t_c indicates the current training time in arbitrary units, e.g., number of training iterations. As always, the training itself is performed using $\theta(t_c)$, but evaluation and other downstream tasks use a weighted average instead, denoted by $\hat{\theta}(t_c)$. This average is typically defined as a sum over the training iterations:

$$\hat{\theta}(t_c) \;=\; \sum_{t=0}^{t_c} p_{t_c}(t)\, \theta(t), \tag{105}$$

where p_{t_c} is a time-dependent *response function* that sums to one, i.e., $\sum_t p_{t_c}(t) = 1$.

Instead of operating with discretized time steps, we simplify the derivation by treating θ, $\hat{\theta}$, and p_{t_c} as continuous functions defined over $t \in \mathbb{R}_{\geq 0}$. A convenient way to generalize Equation 105 to this case is to interpret p_{t_c} as a continuous probability distribution and define $\hat{\theta}(t_c)$ as the expectation of $\theta(t_c)$ with respect to that distribution:

$$\hat{\theta}(t_c) \;=\; \mathbb{E}_{t \sim p_{t_c}(t)}\big[\theta(t)\big]. \tag{106}$$

Considering the definition of $p_{t_c}(t)$, we can express a large class of practically relevant response functions in terms of a *canonical response function* $f(t)$:

$$p_{t_c}(t) \;=\; \begin{cases} f(t) \,/\, g(t_c) & \text{if } 0 \leq t \leq t_c \\ 0 & \text{otherwise} \end{cases}, \tag{107}$$

$$\text{where} \quad g(t_c) \;=\; \int_0^{t_c} f(t)\, \mathrm{d}t. \tag{108}$$

To characterize the properties, e.g., length, of a given response function, we consider its standard distribution statistics:

$$\mu_{t_c} = \mathbb{E}[t] \quad \text{and} \quad \sigma_{t_c} = \sqrt{\text{Var}[t]} \quad \text{for} \quad t \sim p_{t_c}(t). \tag{109}$$

These two quantities have intuitive interpretations: μ_{t_c} indicates the average delay imposed by the response function, while σ_{t_c} correlates with the length of the time period that is averaged over.

C.2. Traditional EMA profile

The standard choice for the response function is the exponential moving average (EMA) where p_{t_c} decays exponentially as t moves farther away from t_c into the past, often parameterized by EMA half-life λ. In the context of Equation 107, we can express such exponential decay as $p_{t_c}(t) = f(t)/g(t_c)$, where

$$f(t) \;=\; \begin{cases} 2^{t/\lambda} & \text{if } t > 0 \\ \frac{\lambda}{\ln 2}\,\delta(t) & \text{otherwise} \end{cases} \tag{110}$$

$$g(t_c) \;=\; \frac{\lambda\, 2^{t_c/\lambda}}{\ln 2}, \tag{111}$$

and $\delta(t)$ is the Dirac delta function.

The second row of Equation 110 highlights an inconvenient aspect about the traditional EMA. The exponential response function is infinite in the sense that it expects to be able to consult historical values of θ infinitely far in the past, even though the training starts at $t = 0$. Consistent with previous work, we thus deposit the probability mass that would otherwise appear at $t < 0$ onto $t = 0$ instead, corresponding to the standard practice of initializing the accumulated EMA weights to network's initial weights.

This implies that unless $\lambda \ll t_c$, the averaged weights $\hat{\theta}(t_c)$ end up receiving a considerable contribution from the initial state $\theta(0)$ that is, by definition, not meaningful for the task that the model is being trained for.

C.3. Tracking the averaged weights during training

In practice, the value of $\hat{\theta}(t_c)$ is computed during training as follows. Suppose that we are currently at time t_c and know the current $\hat{\theta}(t_c)$. We then run one training iteration to arrive at $t_n = t_c + \Delta t$ so that the updated weights are given by $\theta(t_n)$. Here Δt denotes the length of the training step in whatever units are being used for t.

To define $\theta(t)$ for all values of t, we consider it to be a piecewise constant function so that $\theta(t) = \theta(t_n)$ for every $t_c < t \le t_n$. Let us now write the formula for $\hat{\theta}(t_n)$ in terms of Equations 106 and 107:

$$\hat{\theta}(t_n) = \mathbb{E}_{t\sim p_{t_n}(t)}\big[\theta(t)\big] \tag{112}$$

$$= \int_{-\infty}^{\infty} p_{t_n}(t)\,\theta(t)\,\mathrm{d}t \tag{113}$$

$$= \int_0^{t_n} \frac{f(t)}{g(t_n)}\theta(t)\,\mathrm{d}t \tag{114}$$

$$= \int_0^{t_c} \frac{f(t)}{g(t_n)}\theta(t)\,\mathrm{d}t + \int_{t_c}^{t_n} \frac{f(t)}{g(t_n)}\theta(t)\,\mathrm{d}t \tag{115}$$

$$= \underbrace{\frac{g(t_c)}{g(t_n)}}_{=:\,\beta(t_n)} \underbrace{\int_0^{t_c}\frac{f(t)}{g(t_c)}\theta(t)\mathrm{d}t}_{=\,\hat{\theta}(t_c)} + \frac{\theta(t_n)}{g(t_n)} \underbrace{\int_{t_c}^{t_n} f(t)\mathrm{d}t}_{=\,g(t_n)-g(t_c)} \tag{116}$$

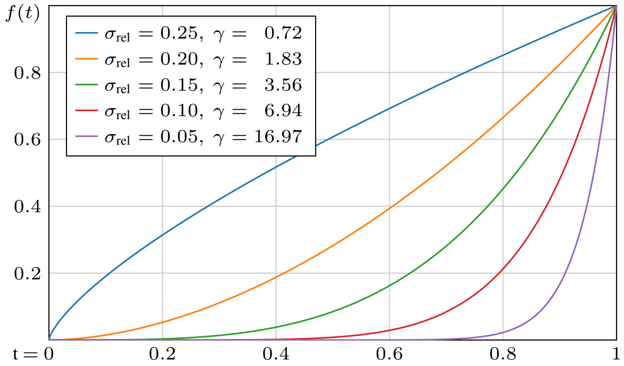


Figure 24. Examples of the canonical response function of our power function EMA profile (Equation 121). Each curve corresponds to a particular choice for the relative standard deviation σ_{rel}; the corresponding exponent γ is calculated using Algorithm 2.

$$= \beta(t_n)\,\hat{\theta}(t_c) + \frac{\theta(t_n)}{g(t_n)}\big(g(t_n) - g(t_c)\big) \tag{117}$$

$$= \beta(t_n)\,\hat{\theta}(t_c) + \Bigg[1 - \underbrace{\frac{g(t_c)}{g(t_n)}}_{=\,\beta(t_n)}\Bigg]\theta(t_n) \tag{118}$$

$$= \beta(t_n)\,\hat{\theta}(t_c) + \big(1 - \beta(t_n)\big)\,\theta(t_n). \tag{119}$$

Thus, after each training iteration, we must linearly interpolate $\hat{\theta}$ toward θ by $\beta(t_n)$. In the case of exponential EMA, $\beta(t_n)$ is constant and, consulting Equation 111, given by

$$\beta(t_n) \;=\; \frac{g(t_c)}{g(t_n)} \;=\; \frac{2^{t_c/\lambda}}{2^{t_n/\lambda}} \;=\; 2^{-\Delta t/\lambda}. \tag{120}$$

C.4. Power function EMA profile

In Section 2, we make two observations that highlight the problematic aspects of the exponential EMA profile. First, it is generally beneficial to employ unconventionally long averages, to the point where $\lambda \ll t_c$ is no longer true. Second, the length of the response function should increase over the course of training proportional to t_c. As such, the definition of $f(t)$ in Equation 110 is not optimal for our purposes.

The most natural requirement for $f(t)$ is that it should be self-similar over different timescales, i.e., $f(ct) \propto f(t)$ for any positive stretching factor c. This implies that the response functions for different values of t_c will also be stretched versions of each other; if t_c doubles, so does σ_{t_c}. Furthermore, we also require that $f(0) = 0$ to avoid meaningless contribution from $\theta(0)$. These requirements are uniquely satisfied, up to constant scaling, by the family of power functions $p_{t_c}(t) = f(t)/g(t_c)$, where

$$f(t) \;=\; t^{\gamma} \quad \text{and} \quad g(t_c) \;=\; \frac{t_c^{\gamma+1}}{\gamma+1}. \tag{121}$$

Algorithm 2 NumPy code for converting σ_{rel} to γ.

```
def sigma_rel_to_gamma(sigma_rel):
    t = sigma_rel ** -2
    gamma = np.roots([1, 7, 16 - t, 12 - t]).real.max()
    return gamma
```

The constant $\gamma > 0$ controls the overall amount of averaging as illustrated in Figure 24.

Considering the distribution statistics of our response function, we notice that p_{t_c} is equal to the beta distribution with $\alpha = \gamma + 1$ and $\beta = 1$, stretched along the t-axis by t_c. The relative mean and standard deviation with respect to t_c are thus given by

$$\mu_{\text{rel}} \;=\; \frac{\mu_{t_c}}{t_c} \;=\; \frac{\gamma+1}{\gamma+2} \tag{122}$$

$$\sigma_{\text{rel}} \;=\; \frac{\sigma_{t_c}}{t_c} \;=\; \sqrt{\frac{\gamma+1}{(\gamma+2)^2\,(\gamma+3)}}. \tag{123}$$

In our experiments, we choose to use σ_{rel} as the primary way of defining and reporting the amount of averaging, including the EDM baseline (CONFIG A) that employs the traditional EMA (Equation 110). Given σ_{rel}, we can obtain the value of γ to be used with Equation 121 by solving a 3rd order polynomial equation and taking the unique positive root

$$\frac{\gamma+1}{(\gamma+2)^2\,(\gamma+3)} \;=\; \sigma_{\text{rel}}^2 \tag{124}$$

$$(\gamma+2)^2\,(\gamma+3) - (\gamma+1)\,\sigma_{\text{rel}}^{-2} \;=\; 0 \tag{125}$$

$$\gamma^3 + 7\gamma^2 + \big(16 - \sigma_{\text{rel}}^{-2}\big)\gamma + \big(12 - \sigma_{\text{rel}}^{-2}\big) \;=\; 0, \tag{126}$$

which can be done using NumPy as shown in Algorithm 2. The requirement $\gamma > 0$ implies that $\sigma_{\text{rel}} < 12^{-0.5} \approx 0.2886$, setting an upper bound for the relative standard deviation.

Finally, to compute $\hat{\theta}$ efficiently during training, we note that the derivation of Equation 119 does not depend on any particular properties of functions f or g. Thus, the update formula remains the same, and we only need to determine $\beta(t_n)$ corresponding to our response function (Equation 121):

$$\beta(t_n) \;=\; \frac{g(t_c)}{g(t_n)} \;=\; \left(\frac{t_c}{t_n}\right)^{\gamma+1} \;=\; \left(1 - \frac{\Delta t}{t_n}\right)^{\gamma+1}. \tag{127}$$

The only practical difference to traditional EMA is thus that $\beta(t_n)$ is no longer constant but depends on t_n.

C.5. Synthesizing novel EMA profiles after training

Using Equation 119, it is possible to track the averaged weights for an arbitrary set of pre-defined EMA profiles during training. However, the number of EMA profiles that can be handled this way is limited in practice by the associated memory and storage costs. Furthermore, it can be challenging to select the correct profiles beforehand, given how much the optimal EMA length tends to vary between different configurations; see Figure 5a, for example. To overcome these challenges, we will now describe a way to synthesize novel EMA profiles *after* the training.

Problem definition. Suppose that we have stored a number of snapshots $\hat{\Theta} = \{\hat{\theta}_1, \hat{\theta}_2, \ldots, \hat{\theta}_N\}$ during training, each of them corresponding to a different response function $p_i(t)$. We can do this, for example, by tracking $\hat{\theta}$ for a couple of different choices of γ (Equation 121) and saving them at regular intervals. In this case, each snapshot $\hat{\theta}_i$ will correspond to a pair (t_i, γ_i) so that $p_i(t) = p_{t_i,\gamma_i}(t)$.

Let $p_r(t)$ denote a novel response function that we wish to synthesize. The corresponding averaged weights are given by Equation 106:

$$\hat{\theta}_r \;=\; \mathbb{E}_{t\sim p_r(t)}\big[\theta(t)\big]. \tag{128}$$

However, we cannot hope to calculate the precise value of $\hat{\theta}_r$ based on $\hat{\Theta}$ alone. Instead, we will approximate it by $\hat{\theta}_r^*$ that we define as a weighted average over the snapshots:

$$\begin{aligned}
\hat{\theta}_r^* \;&=\; \textstyle\sum_i x_i\,\hat{\theta}_i && (129)\\
&=\; \textstyle\sum_i x_i\,\mathbb{E}_{t\sim p_i(t)}\big[\theta(t)\big] && (130)\\
&=\; \textstyle\sum_i x_i \displaystyle\int_{-\infty}^{\infty} p_i(t)\,\theta(t)\,\mathrm{d}t && (131)\\
&=\; \int_{-\infty}^{\infty} \theta(t) \underbrace{\textstyle\sum_i p_i(t)\,x_i}_{=:\,p_r^*(t)}\mathrm{d}t, && (132)
\end{aligned}$$

where the contribution of each $\hat{\theta}_i$ is weighted by $x_i \in \mathbb{R}$, resulting in the corresponding approximate response function $p_r^*(t)$. Our goal is to select $\{x_i\}$ so that $p_r^*(t)$ matches the desired response function $p_r(t)$ as closely as possible.

For notational convenience, we will denote weights by column vector $\mathbf{x} = [x_1, x_2, \ldots, x_N]^\top \in \mathbb{R}^N$ and the snapshot response functions by $\mathbf{p} = [p_1, p_2, \ldots, p_N]$ so that $\mathbf{p}(t)$ maps to the row vector $[p_1(t), p_2(t), \ldots, p_N(t)] \in \mathbb{R}^N$. This allows us to express the approximate response function as an inner product:

$$p_r^*(t) \;=\; \mathbf{p}(t)\,\mathbf{x}. \tag{133}$$

Least-squares solution. To find the value of $\mathbf{x}$, we choose to minimize the L_2 distance between $p_r^*(t)$ and $p_r(t)$:

$$\mathcal{L}(\mathbf{x}) \;=\; \big\|p_r^*(t) - p_r(t)\big\|_2^2 \;=\; \int_{-\infty}^{\infty} \big(p_r^*(t) - p_r(t)\big)^2\,\mathrm{d}t. \tag{134}$$

Let us solve for the minimum of $\mathcal{L}(\mathbf{x})$ by setting its gradient with respect to $\mathbf{x}$ to zero:

$$\mathbf{0} \;=\; \nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x}) \tag{135}$$

$$
\begin{aligned}
&= \nabla_{\mathbf{x}} \left[\int_{-\infty}^{\infty} \big(\mathbf{p}(t)\,\mathbf{x} - p_r(t)\big)^2 \,\mathrm{d}t \right] && (136)\\
&= \int_{-\infty}^{\infty} \nabla_{\mathbf{x}} \Big[\big(\mathbf{p}(t)\,\mathbf{x} - p_r(t)\big)^2 \Big] \,\mathrm{d}t && (137)\\
&= \int_{-\infty}^{\infty} \big(\mathbf{p}(t)\mathbf{x} - p_r(t)\big) \nabla_{\mathbf{x}} \Big[\mathbf{p}(t)\mathbf{x} - p_r(t) \Big] \mathrm{d}t && (138)\\
&= \int_{-\infty}^{\infty} \big(\mathbf{p}(t)\,\mathbf{x} - p_r(t)\big)\, \mathbf{p}(t)^\top \,\mathrm{d}t && (139)\\
&= \int_{-\infty}^{\infty} \big(\mathbf{p}(t)^\top \mathbf{p}(t)\,\mathbf{x} - \mathbf{p}(t)^\top p_r(t)\big) \,\mathrm{d}t && (140)\\
&= \underbrace{\int_{-\infty}^{\infty} \mathbf{p}(t)^\top \mathbf{p}(t) \,\mathrm{d}t}_{=:\,\mathbf{A}} \;\mathbf{x} - \underbrace{\int_{-\infty}^{\infty} \mathbf{p}(t)^\top p_r(t) \,\mathrm{d}t}_{=:\,\mathbf{b}} && (141)
\end{aligned}
$$

where we denote the values of the two integrals by matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ and column vector $\mathbf{b} \in \mathbb{R}^N$, respectively. We are thus faced with a standard matrix equation $\mathbf{A}\mathbf{x} - \mathbf{b} = \mathbf{0}$, from which we obtain the solution $\mathbf{x} = \mathbf{A}^{-1}\,\mathbf{b}$.

Based on Equation 141, we can express the individual elements of $\mathbf{A}$ and $\mathbf{b}$ as inner products between their corresponding response functions:

$$
\begin{aligned}
\mathbf{A} &= [a_{ij}], & a_{ij} &= \langle p_i, p_j \rangle && (142)\\
\mathbf{b} &= [b_i]^\top, & b_i &= \langle p_i, p_r \rangle, && (143)
\end{aligned}
$$

$$
\text{where } \langle f, g \rangle = \int_{-\infty}^{\infty} f(x)\, g(x) \,\mathrm{d}x. \qquad (144)
$$

In practice, these inner products can be computed for arbitrary EMA profiles using standard numerical methods, such as Monte Carlo integration.

Analytical formulas for power function EMA profile. If we assume that $\{p_i\}$ and p_r are all defined according to our power function EMA profile (Equation 121), we can derive an accurate analytical formula for the inner products (Equation 144). Compared to Monte Carlo integration, this leads to a considerably faster and more accurate implementation. In this case, each response function is uniquely defined by its associated (t, γ). In other words, $p_i(t) = p_{t_i,\gamma_i}(t)$ and $p_r(t) = p_{t_r,\gamma_r}(t)$.

Let us consider the inner product between two such response functions, i.e., $\langle p_{t_a,\gamma_a}, p_{t_b,\gamma_b} \rangle$. Without loss of generality, we will assume that $t_a \le t_b$. If this is not the case, we can simply flip their definitions, i.e., $(t_a, \gamma_a) \leftrightarrow (t_b, \gamma_b)$. Now,

$$
\begin{aligned}
&\langle p_{t_a,\gamma_a}, p_{t_b,\gamma_b} \rangle && (145)\\
&\quad= \int_{-\infty}^{\infty} p_{t_a,\gamma_a}(t)\, p_{t_b,\gamma_b}(t) \,\mathrm{d}t && (146)\\
&\quad= \int_{0}^{t_a} \frac{f_{\gamma_a}(t)}{g_{\gamma_a}(t_a)} \cdot \frac{f_{\gamma_b}(t)}{g_{\gamma_b}(t_b)} \,\mathrm{d}t && (147)\\
&\quad= \frac{1}{g_{\gamma_a}(t_a)\, g_{\gamma_b}(t_b)} \int_0^{t_a} f_{\gamma_a}(t)\, f_{\gamma_b}(t) \,\mathrm{d}t && (148)\\
&\quad= \frac{(\gamma_a + 1)\,(\gamma_b + 1)}{t_a^{\gamma_a+1}\, t_b^{\gamma_b+1}} \int_0^{t_a} t^{\gamma_a+\gamma_b} \,\mathrm{d}t && (149)\\
&\quad= \frac{(\gamma_a + 1)\,(\gamma_b + 1)}{t_a^{\gamma_a+1}\, t_b^{\gamma_b+1}} \cdot \frac{t_a^{\gamma_a+\gamma_b+1}}{\gamma_a + \gamma_b + 1} && (150)\\
&\quad= \frac{(\gamma_a + 1)\,(\gamma_b + 1)\,(t_a/t_b)^{\gamma_b}}{(\gamma_a + \gamma_b + 1)\, t_b}. && (151)
\end{aligned}
$$

Note that Equation 151 is numerically robust because the exponentiation by γ_b is done for the ratio t_a/t_b instead of being done directly for either t_a or t_b. If we used Equation 150 instead, we would risk floating point overflows even with 64-bit floating point numbers.

Solving the weights $\{x_i\}$ thus boils down to first populating the elements of $\mathbf{A}$ and $\mathbf{b}$ using Equation 151 and then solving the matrix equation $\mathbf{A}\mathbf{x} = \mathbf{b}$. Algorithm 3 illustrates doing this simultaneously for multiple target response functions using NumPy. It accepts a list of $\{t_i\}$ and $\{\gamma_i\}$, corresponding to the input snapshots, as well as a list of $\{t_r\}$ and $\{\gamma_r\}$, corresponding to the desired target responses. The return value is a matrix whose columns represent the targets while the rows represent the snapshots.

Practical considerations. In all of our training runs, we track two weighted averages $\hat{\theta}_1$ and $\hat{\theta}_2$ that correspond to $\sigma_{\text{rel}} = 0.05$ and $\sigma_{\text{rel}} = 0.10$, respectively. We take a snapshot of each average once every 8 million training images, i.e., between 4096 training iterations with batch size 2048, and store it using 16-bit floating point to conserve disk space. The duration of our training runs ranges between 671–2147 million training images, and thus the number of pre-averaged models stored in the snapshots ranges between 160–512. We find that these choices lead to nearly perfect reconstruction in the range $\sigma_{\text{rel}} \in [0.015, 0.250]$. Detailed study of the associated cost vs. accuracy tradeoffs is left as future work.

Algorithm 3 NumPy code for solving post-hoc EMA weights.

```
def p_dot_p(t_a, gamma_a, t_b, gamma_b):
    t_ratio = t_a / t_b
    t_exp = np.where(t_a < t_b, gamma_b, -gamma_a)
    t_max = np.maximum(t_a, t_b)
    num = (gamma_a + 1) * (gamma_b + 1) * t_ratio ** t_exp
    den = (gamma_a + gamma_b + 1) * t_max
    return num / den

def solve_weights(t_i, gamma_i, t_r, gamma_r):
    rv = lambda x: np.float64(x).reshape(-1, 1)
    cv = lambda x: np.float64(x).reshape(1, -1)
    A = p_dot_p(rv(t_i), rv(gamma_i), cv(t_i), cv(gamma_i))
    B = p_dot_p(rv(t_i), rv(gamma_i), cv(t_r), cv(gamma_r))
    X = np.linalg.solve(A, B)
    return X
```

D. Implementation details

We implemented our techniques on top of the publicly available EDM [17] codebase.[6] We performed our experiments on NVIDIA A100-SXM4-80GB GPUs using Python 3.9.16, PyTorch 2.0.0, CUDA 11.8, and CuDNN 8.9.4. We used 32 GPUs (4 DGX A100 nodes) for each training run, and 8 GPUs (1 node) for each evaluation run.

Table 6 lists the full details of our main models featured in Table 2 and Table 3. Our implementation and pre-trained models are available at `https://github.com/NVlabs/edm2`

D.1. Sampling

We used the 2nd order deterministic sampler from EDM (i.e., Algorithm 1 in [17]) in all experiments with $\sigma(t) = t$ and $s(t) = 1$. We used the default settings $\sigma_{\text{min}} = 0.002$, $\sigma_{\text{max}} = 80$, and $\rho = 7$. While we did not perform extensive sweeps over the number of sampling steps N, we found $N = 32$ to yield sufficiently high-quality results for both ImageNet-512 and ImageNet-64.

In terms of guidance, we follow the convention used by Imagen [35]. Concretely, we define a new denoiser $\hat{D}$ based on the primary conditional model D_θ and a secondary unconditional model D_u:

$$\hat{D}(\boldsymbol{x}; \sigma, \mathbf{c}) \;=\; w\, D_\theta(\boldsymbol{x}; \sigma, \mathbf{c}) + (1 - w)\, D_u(\boldsymbol{x}; \sigma), \quad (152)$$

where w is the guidance weight. Setting $w = 1$ disables guidance, i.e., $\hat{D} = D_\theta$, while increasing $w > 1$ strengthens the effect. The corresponding ODE is then given by

$$\mathrm{d}\boldsymbol{x} \;=\; \frac{\boldsymbol{x} - \hat{D}(\boldsymbol{x}; \sigma, \mathbf{c})}{\sigma}\, \mathrm{d}\sigma. \quad (153)$$

In Table 2 and Table 3, we define NFE as the total number of times that $\hat{D}$ is evaluated during sampling. In other words, we do not consider the number of model evaluations to be affected by the choice of w.

D.2. Mixed-precision training

In order to utilize the high-performance tensor cores available in NVIDIA Ampere GPUs, we use mixed-precision training in all of our training runs. Concretely, we store all trainable parameters as 32-bit floating point (FP32) but temporarily cast them to 16-bit floating point (FP16) before evaluating the model. We store and process all activation tensors as FP16, except for the embedding network and the associated per-block linear layers, where we opt for FP32 due to the low computational cost. In CONFIGS A–B, our baseline architecture uses FP32 in the self-attention blocks as well, as explained in Appendix B.1.

We have found that our models train with FP16 just as well as with FP32, as long as the loss function is scaled with an appropriate constant (see "Loss scaling" in Figures 16–22). In some rare cases, however, we have encountered occasional FP16 overflows that can lead to a collapse in the training dynamics unless they are properly dealt with. As a safety measure, we force the gradients computed in each training iteration to be finite by replacing `NaN` and `Inf` values with 0. We also clamp the activation tensors to range $[-256, +256]$ at the end of each encoder and decoder block. This range is large enough to contain all practically relevant variation (see Figure 15).

D.3. Training data

We preprocess the ImageNet dataset exactly as in the ADM implementation[7] by Dhariwal and Nichol [7] to ensure a fair comparison. The training images are mostly non-square at varying resolutions. To obtain image data in square aspect ratio at the desired training resolution, the raw images are processed as follows:

1. Resize the shorter edge to the desired training resolution using bicubic interpolation.
2. Center crop.

During training, we do not use horizontal flips or any other kinds of data augmentation.

D.4. FID calculation

We calculate FID [12] following the protocol used in EDM [17]: We use 50,000 generated images and all available real images, without any augmentation such as horizontal flips. To reduce the impact of random variation, typically in the order of ±2%, we compute FID three times in each experiment and report the minimum. The shaded regions in FID plots show the range of variation among the three evaluations.

We use the pre-trained Inception-v3 model[8] provided with StyleGAN3 [16], which is a direct PyTorch translation of the original TensorFlow-based model.[9]

D.5. Model complexity estimation

Model complexity (Gflops) was estimated using a PyTorch script that runs the model through `torch.jit.trace` to collect the exact tensor operations used in model evaluation. This list of `aten::*` ops and tensor input and output sizes was run through an estimator that outputs the number of floating point operations required for a single evaluation of the model.

[6] `https://github.com/NVlabs/edm`

[7] `https://github.com/openai/guided-diffusion/blob/22e0df8183507e13a7813f8d38d51b072ca1e67c/guided_diffusion/image_datasets.py#L126`

[8] `https://api.ngc.nvidia.com/v2/models/nvidia/research/stylegan3/versions/1/files/metrics/inception-2015-12-05.pkl`

[9] `http://download.tensorflow.org/models/image/imagenet/inception-2015-12-05.tgz`

Model details	ImageNet-512						ImageNet-64			
	XS	S	M	L	XL	XXL	S	M	L	XL
Number of GPUs	32	32	32	32	32	32	32	32	32	32
Minibatch size	2048	2048	2048	2048	2048	2048	2048	2048	2048	2048
Duration (Mimg)	2147.5	2147.5	2147.5	1879.0	1342.2	939.5	1073.7	2147.5	1073.7	671.1
Channel multiplier	128	192	256	320	384	448	192	256	320	384
Dropout probability	0%	0%	10%	10%	10%	10%	0%	10%	10%	10%
Learning rate max (α_{ref})	0.0120	0.0100	0.0090	0.0080	0.0070	0.0065	0.0100	0.0090	0.0080	0.0070
Learning rate decay (t_{ref})	70000	70000	70000	70000	70000	70000	35000	35000	35000	35000
Noise distribution mean (P_{mean})	-0.4	-0.4	-0.4	-0.4	-0.4	-0.4	-0.8	-0.8	-0.8	-0.8
Noise distribution std. (P_{std})	1.0	1.0	1.0	1.0	1.0	1.0	1.6	1.6	1.6	1.6
Model size and training cost										
Model capacity (Mparams)	124.7	280.2	497.8	777.5	1119.3	1523.2	280.2	497.8	777.5	1119.3
Model complexity (gigaflops)	45.5	102.2	180.8	282.2	405.9	552.1	101.9	180.8	282.1	405.9
Training cost (zettaflops)	0.29	0.66	1.16	1.59	1.63	1.56	0.33	1.16	0.91	0.82
Training speed (images/sec)	8265	4717	3205	2137	1597	1189	4808	3185	2155	1597
Training time (days)	3.0	5.3	7.8	10.2	9.7	9.1	2.6	7.8	5.8	4.9
Training energy (MWh)	1.2	2.2	3.2	4.2	4.0	3.8	1.1	3.2	2.4	2.0
Sampling without guidance, FID										
FID	3.53	2.56	2.25	2.06	1.96	1.91	1.58	1.43	1.33	1.33
EMA length (σ_{rel})	0.135	0.130	0.100	0.085	0.085	0.070	0.075	0.060	0.040	0.040
Sampling cost (teraflops)	4.13	7.70	12.65	19.04	26.83	36.04	6.42	11.39	17.77	25.57
Sampling speed (images/sec/GPU)	8.9	6.4	4.8	3.7	2.9	2.3	10.1	6.6	4.6	3.5
Sampling energy (mWh/image)	17	23	31	41	51	65	15	22	32	43
Sampling with guidance, FID										
FID	2.91	2.23	2.01	1.88	1.85	1.81	–	–	–	–
EMA length (σ_{rel})	0.045	0.025	0.030	0.015	0.020	0.015	–	–	–	–
Guidance strength	1.4	1.4	1.2	1.2	1.2	1.2	–	–	–	–
Sampling cost (teraflops)	6.99	10.57	15.52	21.91	29.70	38.91	–	–	–	–
Sampling speed (images/sec/GPU)	6.0	4.7	3.8	3.0	2.5	2.0	–	–	–	–
Sampling energy (mWh/image)	25	32	39	49	59	73	–	–	–	–
Sampling without guidance, $\text{FD}_{\text{DINOv2}}$										
$\text{FD}_{\text{DINOv2}}$	103.39	68.64	58.44	52.25	45.96	42.84	–	–	–	–
EMA length (σ_{rel})	0.200	0.190	0.155	0.155	0.155	0.150	–	–	–	–
Sampling with guidance, $\text{FD}_{\text{DINOv2}}$										
$\text{FD}_{\text{DINOv2}}$	79.94	52.32	41.98	38.20	35.67	33.09	–	–	–	–
EMA length (σ_{rel})	0.150	0.085	0.015	0.035	0.030	0.015	–	–	–	–
Guidance strength	1.7	1.9	2.0	1.7	1.7	1.7	–	–	–	–

Table 6. Details of all models discussed in Section 4. For ImageNet-512, EDM2-S is the same as CONFIG G in Figure 22.

In practice, a small set of operations dominate the cost of evaluating a model. In the case of our largest (XXL) ImageNet-512 model, the topmost gigaflops producing ops are distributed as follows:

• `aten::_convolution`	545.50 Gflops
• `aten::mul`	1.68 Gflops
• `aten::div`	1.62 Gflops
• `aten::linalg_vector_norm`	1.54 Gflops
• `aten::matmul`	1.43 Gflops

Where available, results for previous work listed in Table 2 were obtained from their respective publications. In cases where model complexity was not publicly available, we used our PyTorch estimator to compute a best effort estimate. We believe our estimations are accurate to within 10% accuracy.

D.6. Per-layer sensitivity to EMA length

List of layers included in the sweeps of Figure 5b in the main paper are listed below. The analysis only includes weight tensors — not biases, group norm scale factors, or affine layers' learned gains.

- `enc-64x64-block0-affine`
- `enc-64x64-block0-conv0`
- `enc-64x64-block0-conv1`
- `enc-64x64-conv`
- `enc-32x32-block0-conv0`
- `enc-32x32-block0-skip`
- `enc-16x16-block0-affine`
- `enc-16x16-block0-conv0`
- `enc-16x16-block2-conv0`
- `enc-8x8-block0-affine`
- `enc-8x8-block0-skip`
- `enc-8x8-block1-conv0`
- `enc-8x8-block2-conv0`
- `dec-8x8-block0-conv0`
- `dec-8x8-block2-skip`
- `dec-8x8-in0-affine`

- `dec-16x16-block0-affine`
- `dec-16x16-block0-conv1`
- `dec-16x16-block0-skip`
- `dec-32x32-block0-conv1`
- `dec-32x32-block0-skip`
- `dec-32x32-up-affine`
- `dec-64x64-block0-conv1`
- `dec-64x64-block0-skip`
- `dec-64x64-block3-skip`
- `dec-64x64-up-affine`
- `map-label`
- `map-layer0`

E. Negative societal impacts

Large-scale image generators such as DALL·E 3, Stable Diffusion XL, or MidJourney can have various negative societal effects, including types of disinformation or emphasizing sterotypes and harmful biases [27]. Our advances to the result quality can potentially further amplify some of these issues. Even with our efficiency improvements, the training and sampling of diffusion models continue to require a lot of electricity, potentially contributing to wider issues such as climate change.

References

[1] Devansh Arpit, Yingbo Zhou, Bhargava Kota, and Venu Govindaraju. Normalization propagation: A parametric technique for removing internal covariate shift in deep networks. In *Proc. ICML*, 2016. 17

[2] Jeremy Bernstein, Arash Vahdat, Yisong Yue, and Ming-Yu Liu. On the distance between two neural networks and the stability of learning. In *Proc. NIPS*, 2020. 17

[3] Jeremy Bernstein, Jiawei Zhao, Markus Meister, Ming-Yu Liu, Anima Anandkumar, and Yisong Yue. Learning compositional functions via multiplicative weight updates. In *Proc. NeurIPS*, 2020. 17

[4] Mikołaj Bińkowski, Danica J. Sutherland, Michael Arbel, and Arthur Gretton. Demystifying MMD GANs. In *Proc. ICLR*, 2018. 1

[5] Andrew Brock, Soham De, Samuel L. Smith, and Karen Simonyan. High-performance large-scale image recognition without normalization. In *Proc. ICML*, 2021. 17

[6] Minhyung Cho and Jaehyung Lee. Riemannian approach to batch normalization. In *Proc. NIPS*, 2017. 17

[7] Prafulla Dhariwal and Alex Nichol. Diffusion models beat GANs on image synthesis. In *Proc. NeurIPS*, 2021. 4, 5, 24

[8] Spyros Gidaris and Nikos Komodakis. Dynamic few-shot visual learning without forgetting. In *Proc. CVPR*, 2018. 12

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *Proc. ICCV*, 2015. 7

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *Proc. ECCV*, 2016. 5

[11] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (GELUs). *CoRR*, abs/1606.08415, 2016. 5

[12] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. GANs trained by a two time-scale update rule converge to a local Nash equilibrium. In *Proc. NIPS*, 2017. 24

[13] Elad Hoffer, Ron Banner, Itay Golan, and Daniel Soudry. Norm matters: Efficient and accurate normalization schemes in deep networks. In *Proc. NIPS*, 2018. 17

[14] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of GANs for improved quality, stability, and variation. In *Proc. ICLR*, 2018. 12, 14

[15] Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Analyzing and improving the image quality of StyleGAN. In *Proc. CVPR*, 2020. 17

[16] Tero Karras, Miika Aittala, Samuli Laine, Erik Härkönen, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Alias-free generative adversarial networks. In *Proc. NeurIPS*, 2021. 17, 24

[17] Tero Karras, Miika Aittala, Timo Aila, and Samuli Laine. Elucidating the design space of diffusion-based generative models. In *proc. NeurIPS*, 2022. 4, 7, 8, 24

[18] Alex Kendall, Yarin Gal, and Roberto Cipolla. Multi-task learning using uncertainty to weigh losses for scene geometry and semantics. In *Proc. CVPR*, 2018. 9

[19] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proc. ICLR*, 2015. 14, 16

[20] Daniel Kunin, Javier Sagastuy-Brena, Surya Ganguli, Daniel L. K. Yamins, and Hidenori Tanaka. Neural mechanics: Symmetry and broken conservation laws in deep learning dynamics. In *Proc. ICLR*, 2021. 17

[21] Tuomas Kynkäänniemi, Tero Karras, Samuli Laine, Jaakko Lehtinen, and Timo Aila. Improved precision and recall metric for assessing generative models. In *Proc. NeurIPS*, 2019. 1

[22] Twan van Laarhoven. L_2 regularization versus batch and weight normalization. *CoRR*, abs/1706.05350, 2017. 16, 17

[23] Zhiyuan Li and Sanjeev Arora. An exponential learning rate schedule for deep learning. In *Proc. ICLR*, 2020.

[24] Zhiyuan Li, Kaifeng Lyu, and Sanjeev Arora. Reconciling modern deep learning with traditional optimization analyses: The intrinsic learning rate. In *Proc. NeurIPS*, 2020. 17

[25] Yang Liu, Jeremy Bernstein, Markus Meister, and Yisong Yue. Learning by turning: Neural architecture aware optimisation. In *Proc. ICML*, 2021. 17

[26] Chunjie Luo, Jianfeng Zhan, Xiaohe Xue, Lei Wang, Rui Ren, and Qiang Yang. Cosine normalization: Using cosine similarity instead of dot product in neural networks. In *Proc. ICANN*, 2018. 12

[27] Pamela Mishkin, Lama Ahmad, Miles Brundage, Gretchen Krueger, and Girish Sastry. DALL·E 2 preview – risks and limitations. *OpenAI*, 2022. 26

[28] Quang-Huy Nguyen, Cuong Q. Nguyen, Dung D. Le, and Hieu H. Pham. Enhancing few-shot image classification with cosine transformer. *IEEE Access*, 11, 2023. 12

[29] Maxime Oquab, Timothée Darcet, Théo Moutakanni, Huy Vo, Marc Szafraniec, Vasil Khalidov, Pierre Fernandez, Daniel Haziza, Francisco Massa, Alaaeldin El-Nouby, Mahmoud Assran, Nicolas Ballas, Wojciech Galuba, Russell Howes,

Po-Yao Huang, Shang-Wen Li, Ishan Misra, Michael Rabbat, Vasu Sharma, Gabriel Synnaeve, Hu Xu, Hervé Jegou, Julien Mairal, Patrick Labatut, Armand Joulin, and Piotr Bojanowski. DINOv2: Learning robust visual features without supervision. *CoRR*, abs/2304.07193, 2023. 1

[30] William Peebles and Saining Xie. Scalable diffusion models with transformers. In *Proc. ICCV*, 2023. 5

[31] Siyuan Qiao, Huiyu Wang, Chenxi Liu, Wei Shen, and Alan Yuille. Micro-batch training with batch-channel normalization and weight standardization. *CoRR*, abs/1903.10520, 2019. 15

[32] Simon Roburin, Yann de Mont-Marin, Andrei Bursuc, Renaud Marlet, Patrick Pérez, and Mathieu Aubry. Spherical perspective on learning with normalization layers. *Neurocomputing*, 487, 2022. 17

[33] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proc. CVPR*, 2022. 4

[34] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional networks for biomedical image segmentation. In *Proc. MICCAI*, 2015. 5

[35] Chitwan Saharia, William Chan, Saurabh Saxena, Lala Li, Jay Whang, Emily Denton, Seyed Kamyar Seyed Ghasemipour, Burcu Karagol Ayan, S. Sara Mahdavi, Rapha Gontijo Lopes, Tim Salimans, Jonathan Ho, David J. Fleet, and Mohammad Norouzi. Photorealistic text-to-image diffusion models with deep language understanding. In *Proc. NeurIPS*, 2022. 24

[36] Tim Salimans and Diederik P. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Proc. NIPS*, 2016. 15, 16

[37] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, Xi Chen, and Xi Chen. Improved techniques for training GANs. In *Proc. NIPS*, 2016. 1

[38] George Stein, Jesse C. Cresswell, Rasa Hosseinzadeh, Yi Sui, Brendan Leigh Ross, Valentin Villecroze, Zhaoyan Liu, Anthony L. Caterini, J. Eric T. Taylor, and Gabriel Loaiza-Ganem. Exposing flaws of generative model evaluation metrics and their unfair treatment of diffusion models. In *Proc. NeurIPS*, 2023. 1, 3

[39] Matthew Tancik, Pratul P. Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan T. Barron, and Ren Ng. Fourier features let networks learn high frequency functions in low dimensional domains. In *Proc. NeurIPS*, 2020. 13

[40] Cristina Vasconcelos, Hugo Larochelle, Vincent Dumoulin, Rob Romijnders, Nicolas Le Roux, and Ross Goroshin. Impact of aliasing on generalization in deep convolutional networks. In *ICCV*, 2021. 17

[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proc. NIPS*, 2017. 6

[42] Ruosi Wan, Zhanxing Zhu, Xiangyu Zhang, and Jian Sun. Spherical motion dynamics: Learning dynamics of normalized neural network using SGD and weight decay. In *Proc. NeurIPS*, 2021. 17

[43] Yang You, Igor Gitman, and Boris Ginsburg. Large batch training of convolutional networks. *CoRR*, abs/1708.03888, 2017. 17

[44] Yang You, Jing Li, Sashank J. Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training BERT in 76 minutes. In *Proc. ICLR*, 2020. 17

[45] Guodong Zhang, Chaoqi Wang, Bowen Xu, and Roger Grosse. Three mechanisms of weight decay regularization. In *Proc. ICLR*, 2019. 17

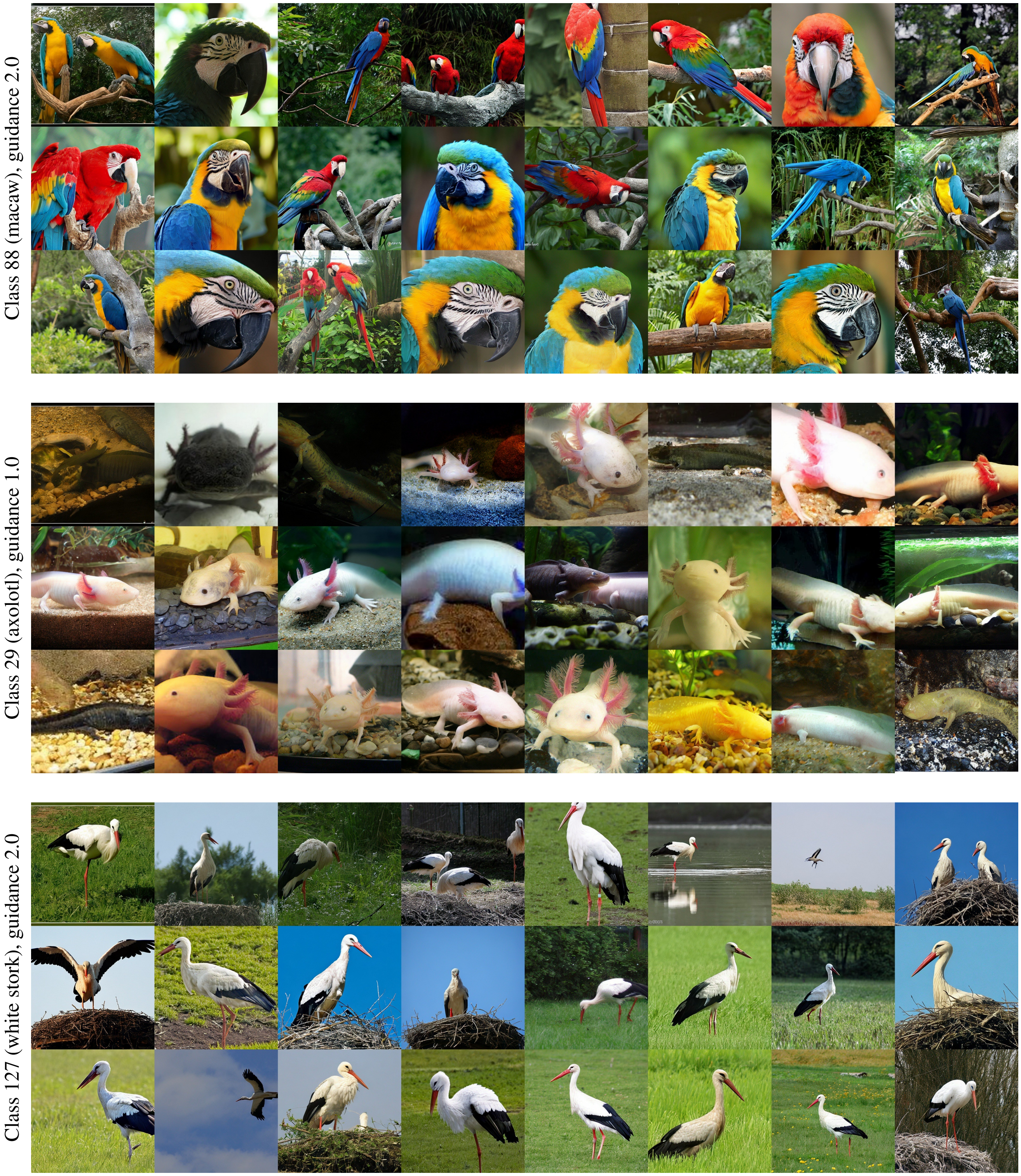

Class 88 (macaw), guidance 2.0
Class 29 (axolotl), guidance 1.0
Class 127 (white stork), guidance 2.0

Figure 25. Uncurated images generated using our largest (XXL) ImageNet-512 model.

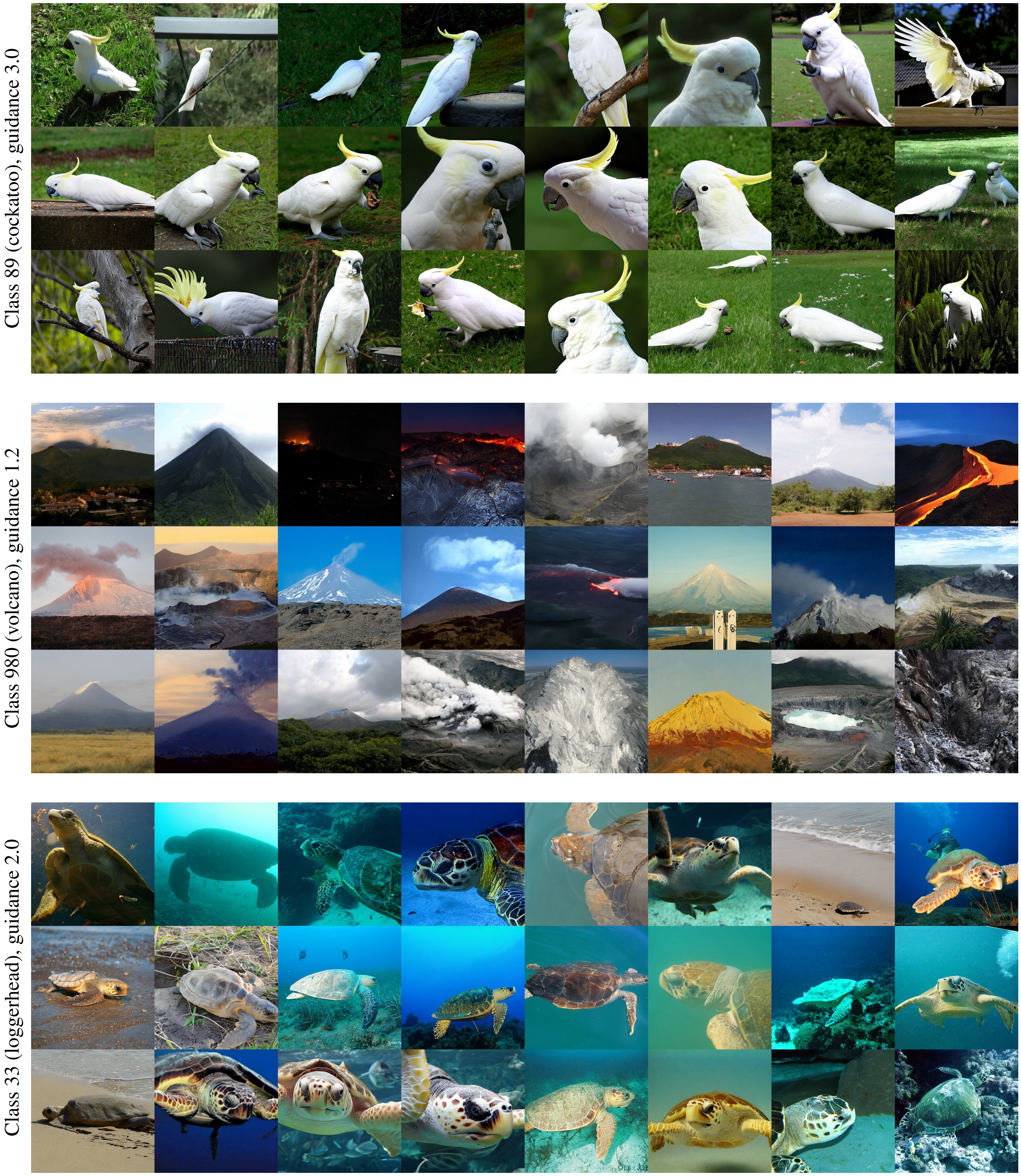

Class 89 (cockatoo), guidance 3.0
Class 980 (volcano), guidance 1.2
Class 33 (loggerhead), guidance 2.0

Figure 26. Uncurated images generated using our largest (XXL) ImageNet-512 model.

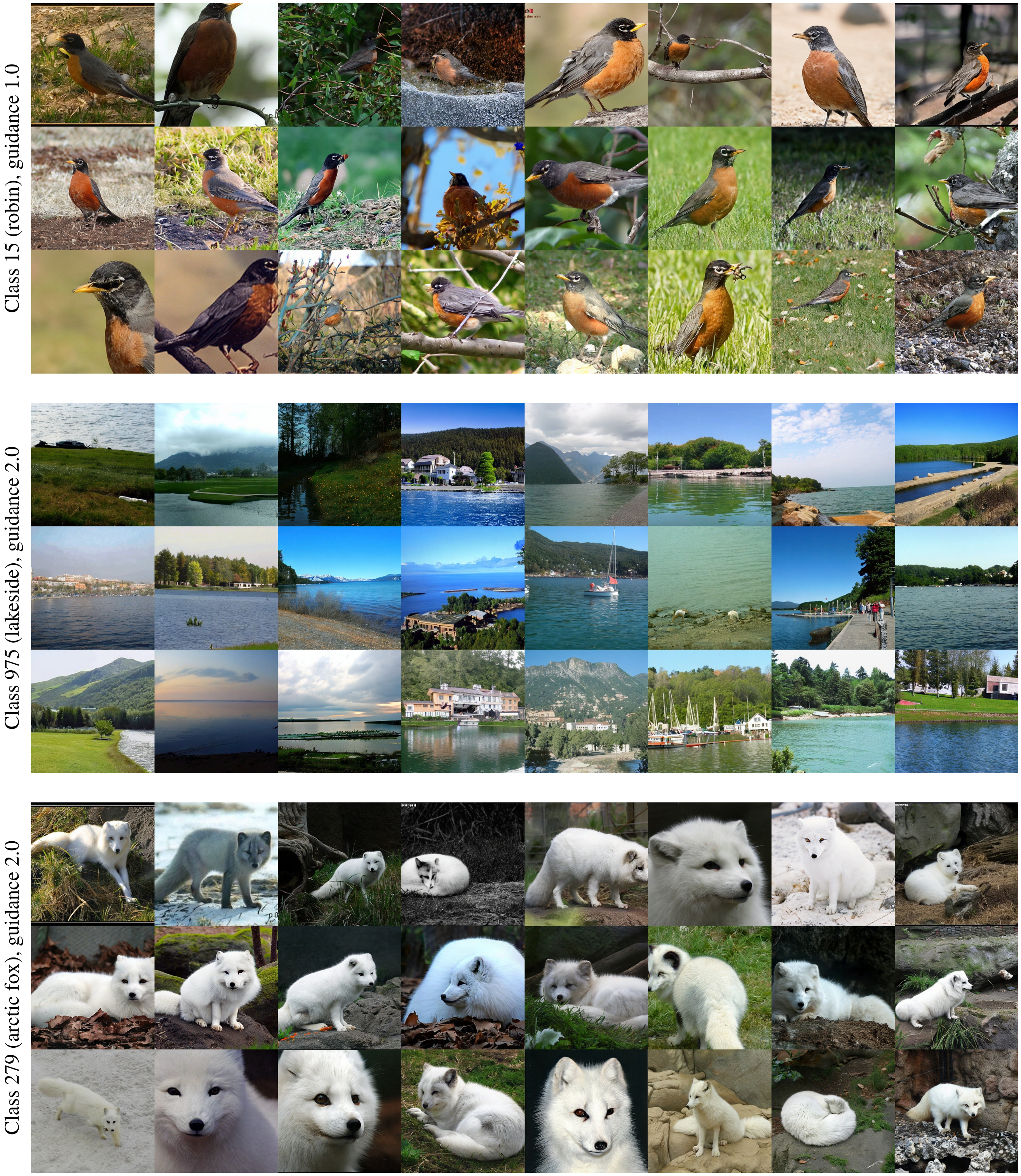

Class 15 (robin), guidance 1.0
Class 975 (lakeside), guidance 2.0
Class 279 (arctic fox), guidance 2.0

Figure 27. Uncurated images generated using our largest (XXL) ImageNet-512 model.