

Encoder-Free Drifting

A generative model that turns noise into an image in one step, explained end to end

Why encoder-free drifting

Drifting trains a generator by comparing two clouds of points, the real samples and the generated ones. It builds a field that points from wherever the generator has put its mass toward wherever the real data sits, and pushes that field to zero. There is no discriminator and no probability to compute, which makes it unusually simple and stable for what it does.

The whole method rests on one question: when are two samples close? Every real and generated sample is weighted by its distance to a probe point, so distance is the only notion of similarity anywhere in the method. Answer that question badly and the field points somewhere unhelpful.

For images, raw pixel distance answers it badly. Two photographs of the same subject, shifted slightly or lit differently, sit far apart in pixel space. Two unrelated photographs that happen to share a dominant colour sit close together. A field built on pixel distances chases the wrong thing.

The usual repair is to measure distances inside a **pretrained encoder** instead. A network already trained on some large labelled collection of images maps pictures into a space where nearby points tend to share content, and distances taken there behave far better. The repair works. It also creates a dependency that is awkward to reason about. The generator inherits whatever that encoder learned, including its training data and its blind spots. Statements about training without supervision become harder to make cleanly. And since the standard image quality scores are themselves computed inside pretrained networks, an objective built on one pretrained feature space and a score built on another share more machinery than is comfortable.

This model asks whether the encoder can be removed from training altogether. Distances are measured either directly in pixels or inside the generator's **own** hidden features, which are learned from scratch alongside everything else. No pretrained network is loaded at any point while the model trains.

What that settles, and what it does not. Evaluation still uses FID, which runs inside a pretrained network, so the pretrained machinery is gone from the training loop while it remains in the measurement. Judging your own samples with your own features is also circular in a way that a fixed external encoder is not, since nothing forces those features to stay meaningful as training proceeds. And the field here is a bounded correction sitting on top of a separate objective, so the experiment speaks to whether a self-supplied similarity measure can help. Whether one could carry the training by itself is a different question, and this work does not answer it.

What the experiment does show is that the substitution earns its place. The same field, computed on raw pixels, made the results worse. Computed in the model's own features, it made them better. That gap is the interesting part, and it suggests the route is worth following further.

Doing it in one step

The second thread is speed. Diffusion models generate gradually, starting from static and removing a little noise at a time, checking their work after every pass. The results are excellent, and a single image can take hundreds of passes through the network.

A **one-step** model does the whole journey in one pass, which is hundreds of times cheaper. The difficulty is that the network can no longer correct itself along the way, so it has to get the entire path from noise to image right on the first attempt. **Mean Flow** showed how to train that: teach the network that one big jump has to agree with the many small steps it replaces.

The model in this document puts the two threads together. A Mean Flow objective does the generative work. Drifting is demoted from being the whole loss to being a bounded correction applied on top of it, with similarity judged as described above. A fourth step, averaging the saved checkpoints, is added at the end and costs nothing.

Part 1: the foundation (EMF)

Mixing. Take a real image and some random noise, pick a number between 0 and 1, and blend them:

$$z_t = (1 - t)x + t\varepsilon$$

x is a real image, ε is random noise of the same shape, and t is the mixing fraction. At $t=0$ you have the clean image, at $t=1$ pure noise, and in between a half-ruined picture.

What the network computes. The network reads a mixture and predicts the clean image behind it. It takes two extra numbers:

$$\hat{x}_\theta(z, t, h), \quad h = t - r$$

θ are the network's weights, t is the current noise level and r is the level being jumped to, so h is the distance travelled. Half of the training examples use $h=0$, which is ordinary supervised denoising and anchors everything else.

The consistency rule. A network that only practised small jumps would still need many calls at generation time, so the remaining examples enforce agreement between scales. First measure how far the network's own answer moves when the jump shrinks slightly:

$$u = \frac{\hat{x}_\theta(z', t - \delta, h - \delta) - \hat{x}_\theta(z, t, h)}{\delta}$$

δ is a small step, 0.001 in this run, and z' is the state after advancing by it. This is a finite difference: it approximates the rate at which the prediction changes as the jump size varies.

Scale that quantity and add it to the clean image to form the target:

$$c(t, r) = (t - r - \delta) \frac{t}{\max(r, 0.1)}$$

$$\text{target} = x + c(t, r) \cdot \text{sg}[u]$$

c weights the correction by how long the jump is and how close its landing point is to clean data. The floor of 0.1 in the denominator keeps it finite as r approaches zero. $\text{sg}[\cdot]$ means *stop gradient*: the value is used as a fixed number, and the network is blocked from adjusting how it was computed. Without that block the network could satisfy the rule by making all of its answers equally uninformative.

The loss is then an ordinary squared error against that target:

$$\mathcal{L}_{\text{EMF}} = \mathbb{E} \|\hat{x}_\theta(z, t, h) - \text{target}\|^2$$

Generating. Draw noise and make a single call at the largest available jump:

$$x = \hat{x}_\theta(\varepsilon, t = 1, h = 1)$$

Part 2: the correction (ASFD)

The foundation learns the general shape of the data but tends to leave gaps, producing plenty of plausible images while quietly skipping certain kinds altogether. The corrections push it toward whatever it is missing.

The field. Weight every sample by how close it is, using a softmax over distances:

$$w_a(z) = \frac{\exp(-\|z - y_a\|/\tau)}{\sum_c \exp(-\|z - y_c\|/\tau)}$$

τ is the bandwidth, which sets how quickly influence falls off with distance. Nearby samples take most of the weight and distant ones get almost none.

Then subtract the weighted average of generated samples from the weighted average of real ones:

$$V_\tau(z) = \sum_a w_a(z) y_a^+ - \sum_b w_b(z) y_b^-$$

y^+ are real samples and y^- are generated ones. The result is an arrow at every point z , pointing away from wherever the generator has piled up too much and toward wherever the real data actually sits.

Driving the length of that arrow to zero everywhere pulls the two clouds together. A single bandwidth sees only one scale, so several run at once:

$$\mathcal{L}_{\text{field}} = \frac{1}{|\mathcal{T}|} \sum_{\tau \in \mathcal{T}} \mathbb{E}_z \|V_\tau(z)\|^2$$

A narrow τ notices fine detail while ignoring the big picture, and a wide τ sees overall shape while missing detail. Squaring each bandwidth separately before summing matters: adding the arrows together first would let two wrong fields cancel each other out.

These distance weights supply the entire notion of similarity, which is why the method needs no pretrained encoder.

The second correction. A different check compares the two distributions through their characteristic functions:

$$\mathcal{L}_{\text{anchor}} = \mathbb{E}_{\omega \sim \rho} \left| \mathbb{E}_p e^{i\langle \omega, X \rangle} - \mathbb{E}_q e^{i\langle \omega, Y \rangle} \right|^2$$

ω is a random frequency drawn from a distribution ρ , with p the real distribution and q the generated one. The exponential is a wave pattern, so the quantity asks whether both sets of samples respond to the same waves the same way on average. If ρ covers all frequencies this reaches zero exactly when $p=q$, which gives the second correction a clean mathematical guarantee that the field on its own lacks.

Why they are capped. Neither correction serves as the loss. Each produces its own gradient, which is shrunk before it is added:

$$s_i = \min \left(1, \frac{c_i \|g_{\text{primary}}\|}{\|g_i\|} \right), \quad g = g_{\text{primary}} + \sum_i s_i g_i$$

g_i is the gradient from correction i and c_i is its cap, which here is 0.15, 0.10 and 0.10. A correction already smaller than its cap passes through untouched, and a larger one is scaled down until it fits. Left uncapped, a correction that is confidently wrong early in training can drag the generator somewhere it never recovers from.

Where similarity is judged. Two of the three corrections are the same field, computed in two different places: once directly on pixels, once on the model's own internal features. The only thing that changes is the space the distances are measured in. On real images the feature version helps and the pixel version hurts, which is what you would expect, since two photographs can share pixel statistics while showing completely different things.

Part 3: averaging the checkpoints

Training never settles into one perfect setting of the weights. Every update is computed from a small random batch, so the weights keep jittering around a good region without ever stopping inside it, and any single checkpoint is a slightly random pick from that region. So save the weights periodically and average them once training ends:

$$\bar{\theta} = \frac{1}{k} \sum_{i=1}^k \theta_i$$

θ_i is the i -th saved checkpoint and k is how many go into the average. What gets averaged here is the weights themselves, and not the images they produce.

Much of the jitter is random, so it cancels, and what survives sits closer to the middle of the good region than any individual checkpoint did. The step costs nothing beyond arithmetic on files you already saved. Generation then runs with the averaged weights:

$$x = \hat{x}_{\bar{\theta}}(\varepsilon, t = 1, h = 1)$$

There is a limit to it. Average in checkpoints from too early and you begin mixing in weights from when the model was genuinely worse, and the result degrades along with them. A moderate window works, and past some width the benefit disappears.

Part 4: the CIFAR-10 experiment

CIFAR-10 is 50,000 small photographs, 32×32 pixels, in ten categories. The generator is a 37.7M-parameter transformer that reads each image as a grid of patches. It is trained *unconditionally*, meaning it is never told which category to produce, and it is evaluated with **FID**, which compares statistics of generated images against real ones. Lower is better.

Three choices mattered more than everything else, and all three repaired something that was already broken. First, the times (t, h) are now drawn evenly, where the original distribution was skewed and rarely visited the setting used for generation. Second, the weight the loss places on extremely noisy examples is capped, which stopped a handful of them from dominating every update. Third, the limit on update size was raised to match the updates actually being produced, since almost every update had been hitting the old limit and flattening the learning rate schedule. Before these repairs the objective was effectively untrainable.

Results

Each stage, measured the same way on 50,000 generated samples:

Stage	FID	What changed
Starting point	83.65	the objective as originally written
+ repaired objective	18.92	time sampling, loss weight, update limit
+ drifting correction	14.98	the self-feature field
+ checkpoint averaging	13.17	no further training at all

The finished model is **6.4× better than where it started** and produces an image in a single network call, with no pretrained network involved anywhere. Repairing the objective was worth the most by a wide margin. The drifting correction added about four FID, and averaging added another 1.8 while requiring no additional training.

The improvement arrived almost entirely as better *coverage*. The model produced a wider variety of images, while the quality of any individual picture stayed roughly where it was. That is what the drifting correction is designed to do, so the gain landed where the design predicted.

Where it could go next

More training and a larger network. The result is limited by architecture and compute. Every structural change tried made it worse, and more training of the same kind did not help, so the remaining headroom lies outside these weights.

Measure the second correction on its own. Every run so far had the frequency-based correction switched on, so its individual contribution has never been isolated. One additional run would settle it.

Watch the samples as well as the score. One configuration was found that produces a small fraction of wildly out-of-range images while its FID still looks healthy, because FID is computed after images are clipped to the valid range. Recording the largest pixel value alongside FID would catch that for free.

Seeing it work on something small

A companion notebook runs this same stack on two-dimensional toy distributions in about three minutes. Because each data point is just two numbers, every stage can be plotted directly. You can watch the foundation cover only part of the target, watch the corrections fill in what it missed, and watch averaging sharpen the result.

The toy version also exposes a limitation that images hide. A one-step generator is a smooth map applied to a blob of noise, and a smooth map cannot tear a connected blob into separate pieces. It can only stretch thin bridges between them. So a target made of eight isolated clusters stays faintly connected however long you train it. In two dimensions there is nowhere to route around that problem. In the thousands of dimensions a real image lives in there is far more room, which is part of why the approach does better on CIFAR-10 than the toy version might suggest.

Recap: from noise to an image

The whole thing in one pass, from the top.

Training, one step. Draw a real image and some random noise, and pick two times. Blend the image and the noise to make the training input. The network looks at that blend and predicts the clean image. The target it is scored against is the clean image plus a correction term that forces long jumps to agree with short ones. Comparing prediction to target gives the first gradient.

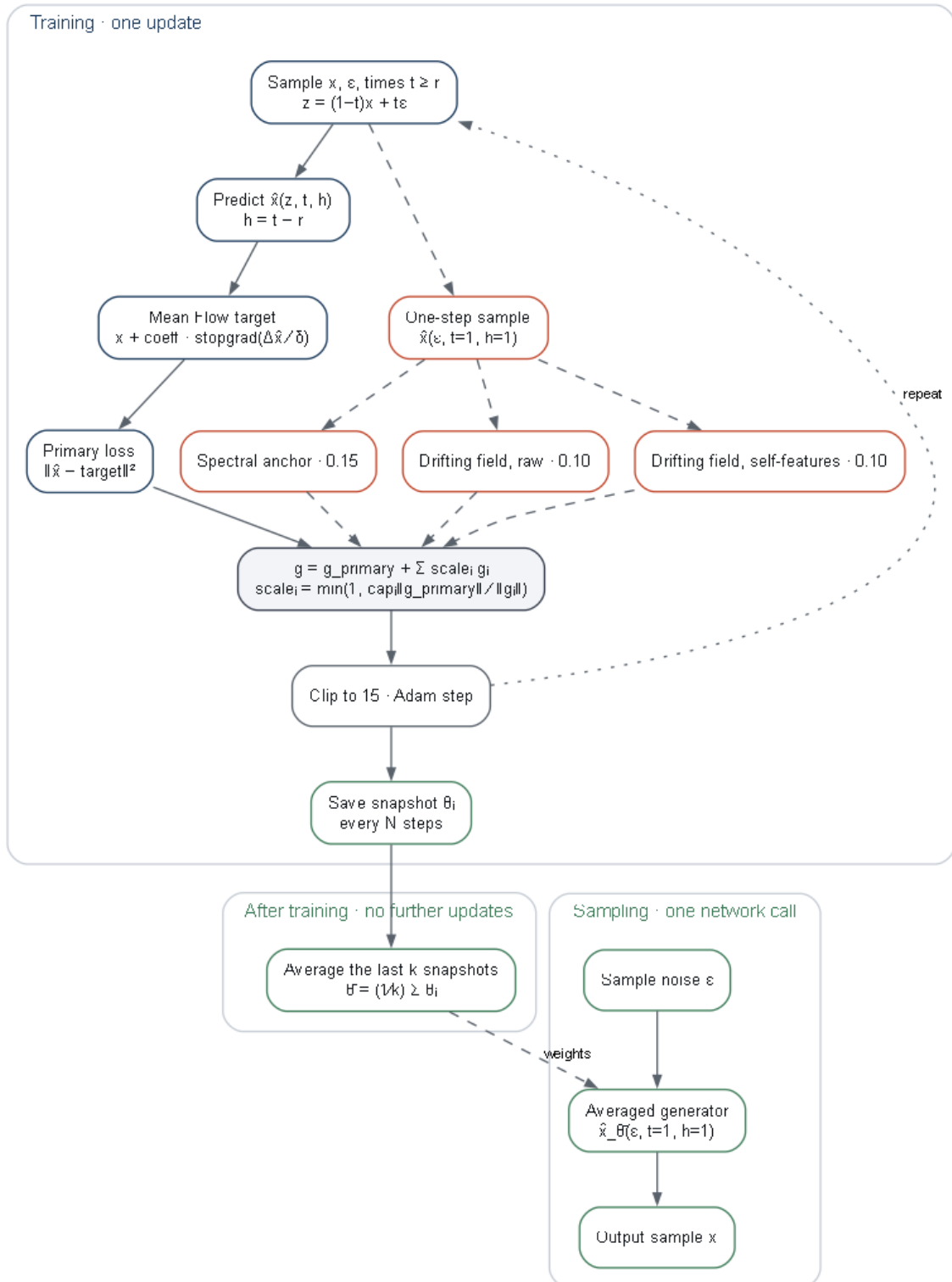
Separately, generate a batch of images from noise, one network call each. Compare them against real samples twice over: once with the drifting field, which points from where the generator put its mass toward where the real data sits, and once with the frequency anchor, which checks whether both sets respond to random wave patterns the same way. Each comparison produces its own gradient. Shrink those so neither can outweigh the first, add everything together, and take a single optimiser step. Every few thousand steps, save a copy of the weights. Then repeat, several hundred thousand times.

After training. Take the last several saved copies and average them into one set of weights. Nothing is trained further, and this step is pure arithmetic on files already on disk.

Generating. Draw a vector of random noise, the same shape as an image. Pass it through the averaged network once, with $t=1$ and $h=1$, which asks for the largest jump the model knows how to make. What comes out is the image. There is no second pass, no noise schedule, and no refinement stage. The samples at the end of this document were each produced that way.

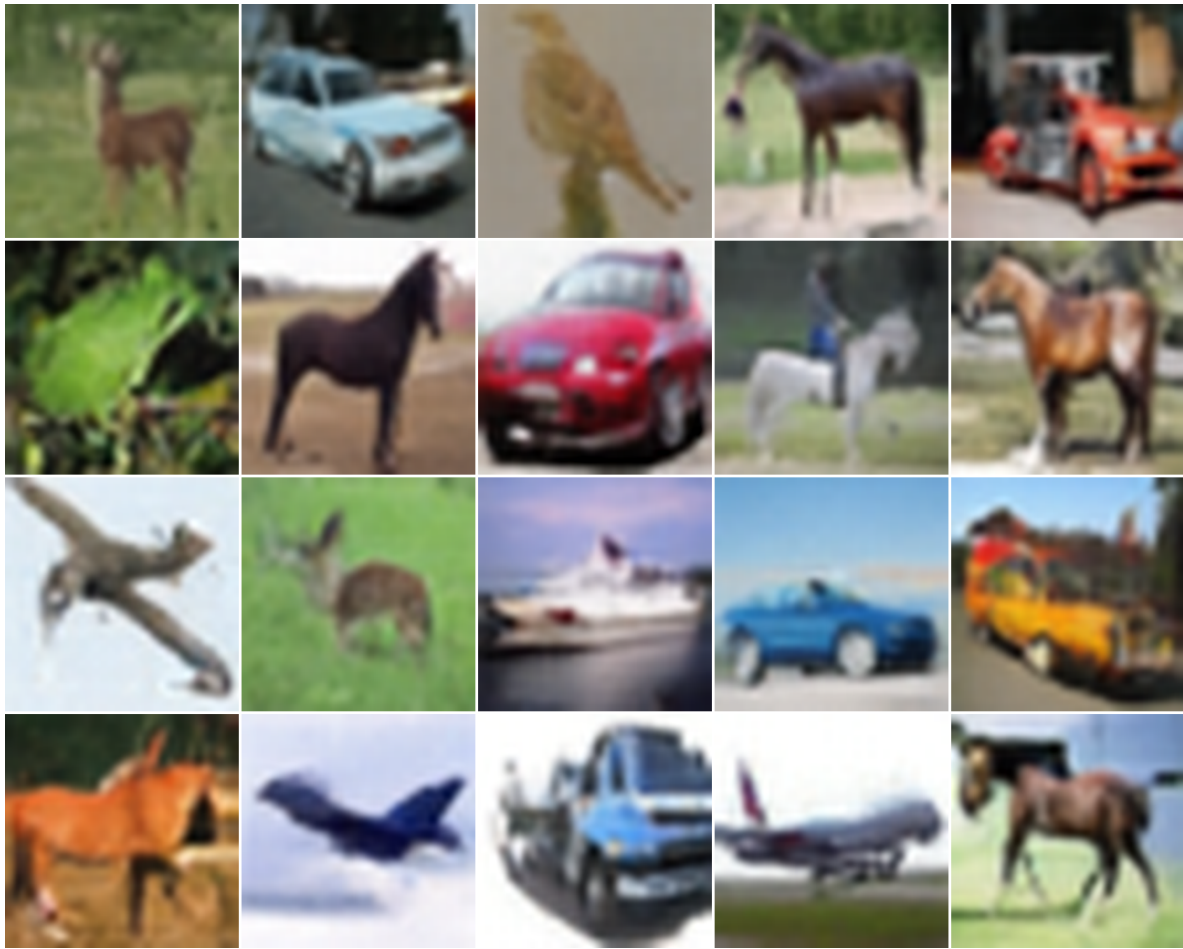
The whole process at a glance

Blue is the main objective, red the three capped corrections, and green the averaging and sampling path. Dashed arrows carry the corrections, which are scaled down before they join the primary gradient.



Samples from the final model

Each image below was produced by a single pass through the network, starting from random noise. The model was never told what to draw.



Twenty samples selected by hand from a batch of 128, chosen to be legible and to span several categories. Typical samples look rougher than these. At 32×32 pixels an FID of 13 buys you recognisable subjects, some way short of clean photographs.

References

Liu, Y. et al. *Euler Mean Flow*. arXiv:2602.02571. <https://arxiv.org/abs/2602.02571>

Deng, Z., Li, X., Li, Y., Du, C. and He, K. *Generative Modeling via Drifting*. arXiv:2602.04770. <https://arxiv.org/abs/2602.04770>

Acknowledgement

The experimental code, the analysis of the results, the accompanying notebook and this document were produced with the assistance of Claude, an AI system made by Anthropic. Every number reported here is an output of the code in this repository, measured through the same sealed evaluation path each time, and the sample images are the measured model's own output.