

GAN: Generative Adversarial Networks

Aayush Bajaj

July 10, 2026

Contents

1	the minimax game	1
2	why the optimum is $p_g = p_{\text{data}}$	2
3	the non-saturating trick	3
4	training pathologies	3
5	wasserstein gans, honestly	3
6	architectures, two lines	4
7	evaluation: fid	4
8	experiment: a 1d gan learns a gaussian	4
9	vs. stable diffusion	6
10	see also	6

a gan trains a generator by making it play a game against a learned critic: the **generator** G maps noise to samples, the **discriminator** D tries to tell those samples from real data, and each improves by exploiting the other's current weakness — density estimation recast as a two-player minimax game (goodfellow et al. 2014, *generative adversarial networks*). the framework is treated in ch. 20 of (Goodfellow, Ian, 2016).

*no likelihood is ever computed;
the discriminator *is* the loss
function, and it is learned*

1 the minimax game

let p_{data} be the data distribution, p_z a fixed noise prior (gaussian, say), and p_g the distribution of $G(z)$, $z \sim p_z$. the value function is

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{\text{data}}} [\log D(x)] + \mathbb{E}_{z \sim p_z} [\log(1 - D(G(z)))] . \quad (1)$$

D is a binary classifier maximising the log-likelihood of “real vs fake”; G minimises the same quantity — it wants its fakes classified real.

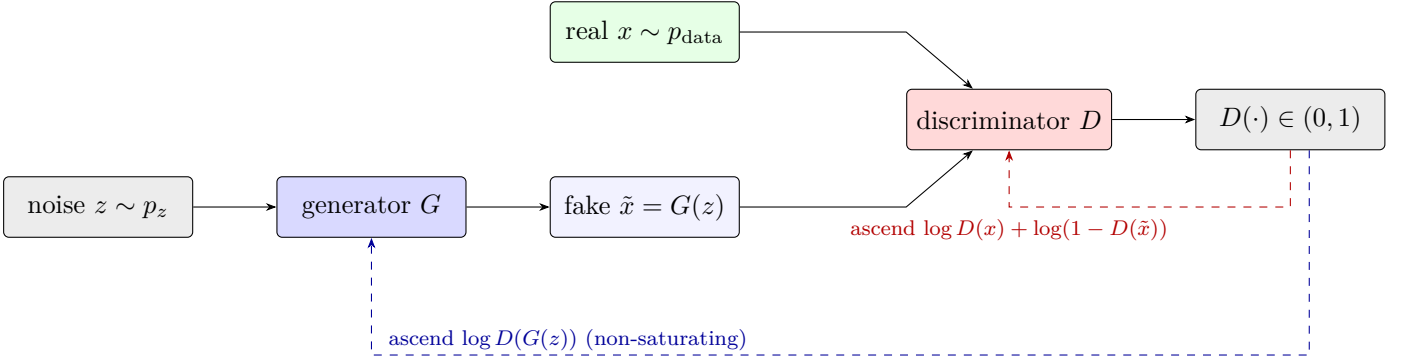


Figure 1: the adversarial loop: the discriminator ascends the value function, the generator descends it through the discriminator’s judgement of its samples.

2 why the optimum is $p_g = p_{\text{data}}$

step 1 — the optimal discriminator. fix G and write the value as an integral:

$$V(D, G) = \int_x \left[p_{\text{data}}(x) \log D(x) + p_g(x) \log (1 - D(x)) \right] dx. \quad (2)$$

the integrand can be maximised pointwise: for constants $a, b \geq 0$ (not both zero), $y \mapsto a \log y + b \log(1 - y)$ peaks at $y^* = \frac{a}{a+b}$.¹ hence

$$D^*(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)}. \quad (3)$$

the perfect discriminator is a *density ratio estimator* — this observation powers half of modern self-supervised learning.

step 2 — the jensen-shannon divergence emerges. substitute D^* back:

$$\begin{aligned} C(G) &= \mathbb{E}_{x \sim p_{\text{data}}} \left[\log \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)} \right] + \mathbb{E}_{x \sim p_g} \left[\log \frac{p_g(x)}{p_{\text{data}}(x) + p_g(x)} \right] \\ &= -\log 4 + \text{KL} \left(p_{\text{data}} \left\| \frac{p_{\text{data}} + p_g}{2} \right\| \right) + \text{KL} \left(p_g \left\| \frac{p_{\text{data}} + p_g}{2} \right\| \right) \\ &= -\log 4 + 2 \text{JSD}(p_{\text{data}} \| p_g), \end{aligned}$$

where the middle line follows by multiplying and dividing each ratio by 2. since $\text{JSD} \geq 0$ with equality iff the arguments coincide, the global minimum of $C(G)$ is $-\log 4$, attained **exactly** when $p_g = p_{\text{data}}$ — at which point $D^* \equiv \frac{1}{2}$: the discriminator is reduced to coin-flipping. the caveats are real, though: the proof is in the space of distributions, assumes D is optimal at every step, and says nothing about whether alternating sgd on network parameters converges there.

¹differentiate: $\frac{a}{y} - \frac{b}{1-y} = 0 \iff a(1-y) = by \iff y = \frac{a}{a+b}$; the second derivative is negative on $(0, 1)$, so it is a maximum.

references

Goodfellow, Ian (2016). *Deep Learning*, MIT Press.

3 the non-saturating trick

early in training G is terrible, D confidently outputs $D(G(z)) \approx 0$, and the generator’s minimax loss $\log(1 - D(G(z)))$ is *flat* there — its gradient vanishes exactly when the generator most needs signal. the standard fix flips the objective: instead of minimising $\log(1 - D(G(z)))$, **maximise** $\log D(G(z))$. near $D(G(z)) = 0$ the new objective has enormous gradient rather than none.

same fixed points, different dynamics: heuristic, but universal since the original paper

4 training pathologies

- **mode collapse.** the generator maps many z to few outputs — it parks all its mass on whatever currently fools D , then hops when D catches up. symptom: low sample diversity; in the 1d experiment below it appears as generated std far below target.
- **vanishing gradients.** if D gets too good and p_g, p_{data} have (near-)disjoint supports — generic for low-dimensional manifolds in high-dimensional pixel space — the js divergence saturates at $\log 2$ and gradients to G die. hence the folklore of carefully rationing discriminator updates.
- **non-convergence.** simultaneous gradient descent-ascent can orbit the equilibrium instead of approaching it (the classic example: $\min_x \max_y xy$ spirals). gans oscillate; losses are nearly useless as progress meters, which is why people monitor samples and fid instead.

5 wasserstein gans, honestly

wgan (arjovsky et al. 2017, *wasserstein gan*) replaces the js divergence with the **earth-mover (wasserstein-1) distance**

$$W(p, q) = \inf_{\gamma \in \Pi(p, q)} \mathbb{E}_{(x, y) \sim \gamma} [\|x - y\|], \quad (4)$$

the minimum cost of transporting the mass of p onto q . the point: when supports are disjoint, js is a constant but W still measures *how far apart* they are — it gives gradients everywhere. by kantorovich-rubinstein duality,

$$W(p_{\text{data}}, p_g) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x \sim p_{\text{data}}} [f(x)] - \mathbb{E}_{x \sim p_g} [f(x)], \quad (5)$$

so the discriminator becomes a **critic** f (no sigmoid, no log) constrained to be 1-lipschitz. enforcing that constraint is the whole engineering problem:

- **weight clipping** (original wgan): clamp every weight to $[-c, c]$. crude — it enforces a lipschitz bound but biases the critic toward simple functions, and c is fiddly. the authors themselves call it “clearly terrible”.
- **gradient penalty** (wgan-gp, gulrajani et al. 2017, *improved training of wasserstein gans*): add $\lambda \mathbb{E}_{\hat{x}} [(\|\nabla_{\hat{x}} f(\hat{x})\| - 1)^2]$ with $\hat{x}$ sampled on lines between real and fake points, exploiting the fact that an optimal critic has unit gradient norm almost everywhere on those lines. much more stable; the standard choice.

honest summary: wgan’s clean theory (a loss that correlates with sample quality, meaningful even for disjoint supports) is bought with an approximation — the critic

is trained finitely many steps (typically 5 per generator step), so you never actually have the supremum, and the duality argument holds only at optimality. it trains more robustly in practice; it is not magic.

6 architectures, two lines

- **dcgan** (radford et al. 2015, *unsupervised representation learning with dcgans*): the recipe that made image gans train at all — all-convolutional G and D , batchnorm, relu/leakyrelu, no pooling. see **cnns** for the building blocks.
- **stylegan** (karras et al. 2019, *a style-based generator architecture*): maps z through an mlp to a style vector that modulates each convolution layer separately, giving coarse-to-fine control and the era's best faces.

7 evaluation: fid

likelihood is unavailable, so evaluation compares feature statistics. **fréchet inception distance** fits gaussians to inception-v3 features of real and generated samples and computes

$$\text{FID} = \|\mu_r - \mu_g\|^2 + \text{Tr}\left(\Sigma_r + \Sigma_g - 2(\Sigma_r \Sigma_g)^{1/2}\right), \quad (6)$$

the *squared* wasserstein-2 (fréchet) distance between the two gaussian fits (heusel et al. 2017, *gans trained by a two time-scale update rule*). lower is better; it punishes both poor fidelity and mode collapse (a collapsed p_g has tiny Σ_g). caveats: biased at small sample counts and inherits inception-v3's imagenet-centric view of what matters in an image.

8 experiment: a 1d gan learns a gaussian

everything above, in ~70 lines of numpy: target $\mathcal{N}(3, 0.5)$, a linear generator $G(z) = Wz + b$ (so p_g is exactly gaussian: learning the target means learning $W \rightarrow \pm 0.5$, $b \rightarrow 3$), a 16-unit tanh mlp discriminator, adam, non-saturating generator loss — backprop written by hand.

```

1  import numpy as np
2
3  rng = np.random.default_rng(0)
4  MU, SIGMA = 3.0, 0.5      # target: N(3, 0.5)
5  G_HIDDEN = 0              # 0 = linear generator; try 16 and watch the std
6                             ↪ collapse
7
8  def mlp_init(n_in, n_h, n_out):
9      if n_h == 0:           # linear model: G(z) = zW + b, so p_g is exactly
10         ↪ gaussian
11         return {"W1": np.ones((n_in, n_out)), "b1": np.zeros(n_out)}
12     return {"W1": rng.normal(0, 0.5, (n_in, n_h)), "b1": np.zeros(n_h),
13             "W2": rng.normal(0, 0.5, (n_h, n_out)), "b2": np.zeros(n_out)}
14
15 def forward(p, x):
16     if "W2" not in p:
17         return None, x @ p["W1"] + p["b1"]
18     h = np.tanh(x @ p["W1"] + p["b1"])
19     return h, h @ p["W2"] + p["b2"]
20
21 def backward(p, x, h, dout):

```

```

20 """grads of scalar loss w.r.t. params and input, given dL/d(output)."""
21 if "W2" not in p:
22     return {"W1": x.T @ dout, "b1": dout.sum(0)}, dout @ p["W1"].T
23 g = {"W2": h.T @ dout, "b2": dout.sum(0)}
24 dh = (dout @ p["W2"].T) * (1 - h**2) # tanh'
25 g["W1"], g["b1"] = x.T @ dh, dh.sum(0)
26 return g, dh @ p["W1"].T
27
28 def adam(p, g, m, v, t, lr=1e-3):
29     for k in p:
30         m[k] = 0.9*m[k] + 0.1*g[k]
31         v[k] = 0.999*v[k] + 0.001*g[k]**2
32         p[k] -= lr * (m[k]/(1-0.9**t)) / (np.sqrt(v[k]/(1-0.999**t)) + 1e-8)
33
34 sigmoid = lambda z: 1/(1+np.exp(-z))
35
36 G, D = mlp_init(1, G_HIDDEN, 1), mlp_init(1, 16, 1)
37 mG = {k: 0.0 for k in G}; vG = {k: 0.0 for k in G}
38 mD = {k: 0.0 for k in D}; vD = {k: 0.0 for k in D}
39 B = 128
40
41 print(f"target: mean={MU}, std={SIGMA}")
42 print(f"{'step':>6} {'gen mean':>9} {'gen std':>8} {'D(real)':>8}
43 ↪ {'D(fake)':>8}")
44 for t in range(1, 10001):
45     # --- discriminator step: ascend log D(x) + log(1 - D(G(z)))
46     x_real = rng.normal(MU, SIGMA, (B, 1))
47     z = rng.normal(0, 1, (B, 1))
48     hG, x_fake = forward(G, z)
49     hr, sr = forward(D, x_real); pr = sigmoid(sr)
50     hf, sf = forward(D, x_fake); pf = sigmoid(sf)
51     gr, _ = backward(D, x_real, hr, (pr - 1)/B) # descend the negative
52     gf, _ = backward(D, x_fake, hf, pf/B)
53     adam(D, {k: gr[k] + gf[k] for k in D}, mD, vD, t)
54
55     # --- generator step (non-saturating): ascend log D(G(z))
56     z = rng.normal(0, 1, (B, 1))
57     hG, x_fake = forward(G, z)
58     hf, sf = forward(D, x_fake); pf = sigmoid(sf)
59     _, dxf = backward(D, x_fake, hf, (pf - 1)/B) # d(-log D)/dx_fake
60     gG, _ = backward(G, z, hG, dxf)
61     adam(G, gG, mG, vG, t)
62
63     if t in (1, 1000, 2000, 4000, 7000, 10000):
64         z = rng.normal(0, 1, (4000, 1))
65         _, xs = forward(G, z)
66         _, sr4 = forward(D, rng.normal(MU, SIGMA, (4000, 1)))
67         _, sf4 = forward(D, xs)
68         print(f"{'t':>6} {'xs.mean():>9.3f} {'xs.std():>8.3f} "
69               f"{'sigmoid(sr4).mean():>8.3f} {'sigmoid(sf4).mean():>8.3f}")

```

target: mean=3.0, std=0.5

step	gen mean	gen std	D(real)	D(fake)
1	-0.004	0.997	0.739	0.498
1000	1.406	0.522	0.885	0.129
2000	2.752	0.000	0.607	0.395
4000	3.018	0.001	0.505	0.492
7000	2.977	0.405	0.505	0.497
10000	3.000	0.508	0.501	0.501

read the last row against the theory: generated mean 3.000 and std 0.508 against targets 3.0 and 0.5, with $D(\text{real}) = D(\text{fake}) = 0.501 \approx \frac{1}{2}$ — the coin-flipping equilibrium $D^* = \frac{1}{2}$ predicted by the optimality proof. the trajectory is also honest

about the dynamics: around steps 2000–4000 the std *collapses to nearly zero* (the generator camps at the target mean) before the discriminator forces it back out — a transient mode collapse and recovery, live.

set `G_HIDDEN = 16` (a tanh-mlp generator) and the same run gets the mean right but the std never recovers:

step	gen mean	gen std	D(real)	D(fake)
1	-0.011	0.462	0.421	0.500
1000	3.180	0.120	0.494	0.495
2000	2.710	0.087	0.503	0.507
4000	3.823	0.074	0.469	0.495
7000	2.714	0.033	0.514	0.512
10000	2.783	0.053	0.516	0.512

that is textbook mode collapse — more generator capacity, worse distribution — and a taste of why gan training earned its reputation.

9 vs. stable diffusion

the modern comparison the stub asked for: [diffusion models](#) replaced gans as the default image generator by trading the adversarial game for a stable regression objective (predict the noise) — no equilibrium to chase, no mode collapse, a usable training loss, at the cost of many network evaluations per sample instead of one. gans survive where single-forward-pass sampling matters (super-resolution, real-time synthesis) and as perceptual losses inside other pipelines — including the adversarial term used to train stable diffusion’s own vae decoder.

10 see also

- [stable diffusion](#) — the successor regime for image generation
- [autoencoders](#) — the other classical route to a latent-variable generator
- [cnn from scratch](#) — the convolutional machinery inside dcgan
- [feedforward networks](#) — backprop, the thing written by hand above