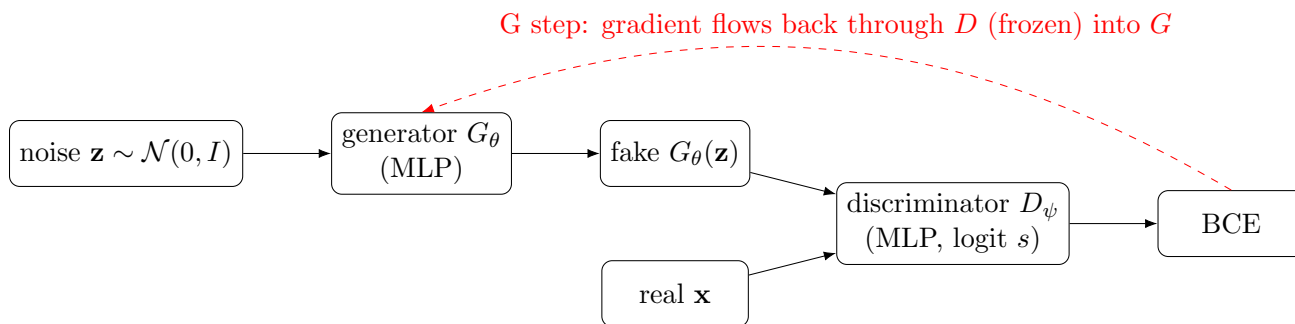


Generative Adversarial Networks: Learning to Sample by Playing a Game

Lecture Notes Consolidated with Claude AI — CS6140 Machine Learning

1 The Big Picture: Two Networks Playing a Game

Before any math, here is the whole method. A GAN has two networks. The **generator** G turns random noise $\mathbf{z}$ into a fake data point $G(\mathbf{z})$. The **discriminator** D looks at a point and outputs the probability that it is real. They are trained against each other, one step each, over and over:



Algorithm: GAN training (one iteration)

1. **D step.** Draw a batch of real $\mathbf{x}$ (label 1) and a batch of fakes $G(\mathbf{z})$ (label 0), with the fakes *detached*: treated as fixed inputs, so this step's gradient does not reach G . Take one optimizer step on ψ to reduce D 's BCE loss (Eq. 1).
2. **G step.** Draw fresh noise, compute $D(G(\mathbf{z}))$, and take one optimizer step on θ only, to make D call these fakes real (the loss in §6). The gradient flows through D 's layers into G 's layers. D 's weights are not updated in this step.
3. Repeat. (Some recipes take $k > 1$ D steps per G step. HW5 Problem 4(A) uses $k = 1$; 4(B) compares $k = 1$, $k = 5$, and 5 G steps per D step.)

The rest of the note explains each box: what G is and why it needs no density (§2); why D is a logistic regression and what the best D looks like (§3); what the whole game is really minimizing (§4); the training loop in detail (§5); why G uses label 1 (§6); and how it fails (§7).

Almost everything here is something you have already built. Table 1 lists the parts of a GAN and where you have seen each one. Only the last three rows are new ideas; everything above them is reused, sometimes with a small change.

Part	Where you have seen it	What changes
Generator G : noise $\rightarrow$ MLP $\rightarrow$ a data point (§2)	The VAE’s generation step: a random code into the decoder. Structurally, HW5 Problem 3’s decoder.	No encoder in front of it; no sigmoid on the output (it outputs a point, not pixel probabilities); LeakyReLU; trained without any likelihood.
Sampling: draw $\mathbf{z} \sim \mathcal{N}(0, I)$, run G	VAE: generating after training. $\mathbf{x} = \mu + \sigma\epsilon$ is a one-layer linear generator.	Nothing.
Discriminator D : MLP $\rightarrow$ logit $s \rightarrow \sigma(s) = P(\text{real})$ (§3)	HW5 Problem 2(C)’s classifier; HW2’s logistic regression.	One output with BCE, instead of 10 with softmax; LeakyReLU; 2-D input.
Training D : real $\rightarrow 1$, fake $\rightarrow 0$, BCE	Supervised binary classification (HW2).	The “fake” class is produced by G , and it changes after every G step.
Best possible D : $D^* = p_{\text{data}}/(p_{\text{data}} + p_g)$ (§3.1)	HW4 Problem 1: Bayes’ rule with equal priors. For two Gaussians it is LDA’s boundary.	The two densities are unknown, so a trained D only approximates D^* .
D ’s loss = log 4 at equilibrium (§4)	The BCE of a classifier that outputs $\frac{1}{2}$ everywhere.	Nothing (arithmetic).
Backprop through $D \circ G$ (§5)	Week 9: the chain rule through stacked layers, as through the VAE’s encoder and decoder.	Nothing.
Update one network per step; detach (§5)	Choosing which parameters an optimizer updates (Week 9).	detach stops the gradient at G ’s output: one line of code.
What the game minimizes: JSD (§4)	Built from the VAE lecture’s KL divergence.	KL of each distribution to their average. Optional depth.
The loss is another network (§5)	NEW	G is never compared with the data directly. Its training signal is a classifier’s opinion, and that classifier keeps learning.
Training is a game, not a minimization (§5, §7)	NEW	Alternating steps, and no single loss that goes down. Success means D ’s loss sits at log 4; the balance between D and G matters; mode collapse.
Flip G ’s label (§6)	NEW (a small trick)	Train G with label 1, because the original loss has almost no gradient while G is still bad.

Table 1: The parts of a GAN. Nine are reused from earlier in the course; three are new.

1.1 Why a Third Approach?

Both generative models so far write down a density $p(\mathbf{x})$:

- HW4’s GMM: $p(\mathbf{x}) = \sum_k \pi_k \mathcal{N}(\mathbf{x}; \mu_k, \Sigma_k)$, maximized exactly by EM.
- The VAE: $p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{x} | \mathbf{z})p(\mathbf{z})d\mathbf{z}$ has no formula, so we maximized a lower bound (the ELBO) and had to build an encoder to make the bound useful.

But to *generate* we never needed $p(\mathbf{x})$. We only needed a way to produce new samples. A **Generative Adversarial Network (GAN)** builds only the sampler, and trains it with a signal that needs no density

at all: a second network that tries to tell real samples from generated ones.

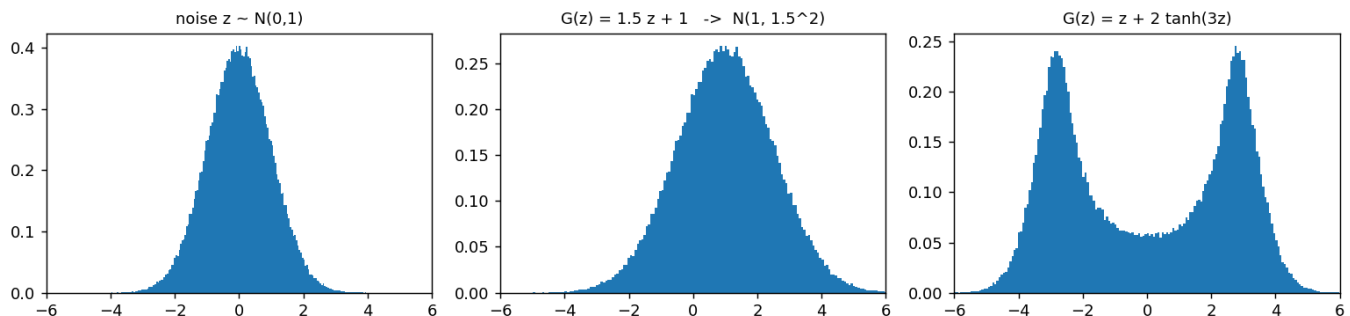
Key point: Picture a counterfeiter G and a detective D . The detective studies real banknotes and counterfeits and learns to tell them apart. The counterfeiter never sees a real banknote. It only learns from the detective’s verdicts which direction would make its fakes more convincing. If both keep improving, the fakes end up indistinguishable from real notes. At no point does anyone write down “the probability of this banknote”.

2 The Generator: the VAE’s Decoder, Without the Encoder

You have seen a generator already. At the end of the VAE lecture, *generating* meant drawing a random code $\mathbf{z} \sim \mathcal{N}(0, I)$ and running the decoder. A GAN’s generator is exactly that kind of network, with no encoder in front of it. What is new is only how it is trained (§5), and the fact that nothing in that training ever needs a formula for the density of its outputs.

Take noise $\mathbf{z} \sim \mathcal{N}(0, I)$ and a network G_θ . Every draw of $\mathbf{z}$ gives a sample $\mathbf{x} = G_\theta(\mathbf{z})$. The distribution of these samples, call it p_g , is the generator’s model.

- **Linear** $G(z) = az + b$: then $x \sim \mathcal{N}(b, a^2)$. This is the VAE lecture’s reparameterization $x = \mu + \sigma\epsilon$, read as a generator.
- **Nonlinear** G : it can stretch some regions of z and squeeze others. Where G is steep, nearby z ’s are spread far apart and few samples land there. Example: $G(z) = z + 2 \tanh(3z)$ is steep near $z = 0$, so it turns one bump into two:



Why not just do maximum likelihood on G ? To evaluate $p_g(\mathbf{x})$ at a data point we would need to know which $\mathbf{z}$ ’s map to $\mathbf{x}$ and how much G stretches space there, i.e. to invert G . A multi-layer network has no formula for that. (And when the noise has fewer dimensions than the data, as for images, the samples live on a thin surface inside data space and $p_g(\mathbf{x})$ is not even an ordinary density.) A GAN is **implicit**: sampling is easy, and the density is never available.

Structure. The generator is the VAE’s decoder with no encoder in front of it: a Week 9 stack of Linear + activation layers, fed pure noise. For real-valued data (such as HW5’s 2-D points) the last layer has *no* squashing activation, because it outputs data values, not probabilities.

3 The Discriminator: a Classifier You Have Already Built

Structurally, D is HW5 Problem 2(C)’s classifier with a single output, and its last layer followed by a sigmoid is HW2’s logistic regression. Its best possible version (§3.1 below) is HW4’s Bayes classifier. Nothing in this section is a new tool; it is logistic regression, pointed at a new question: “real or generated?”

Figure 1 shows both networks layer by layer, at HW5’s sizes, before the math of this section: D is the right half.

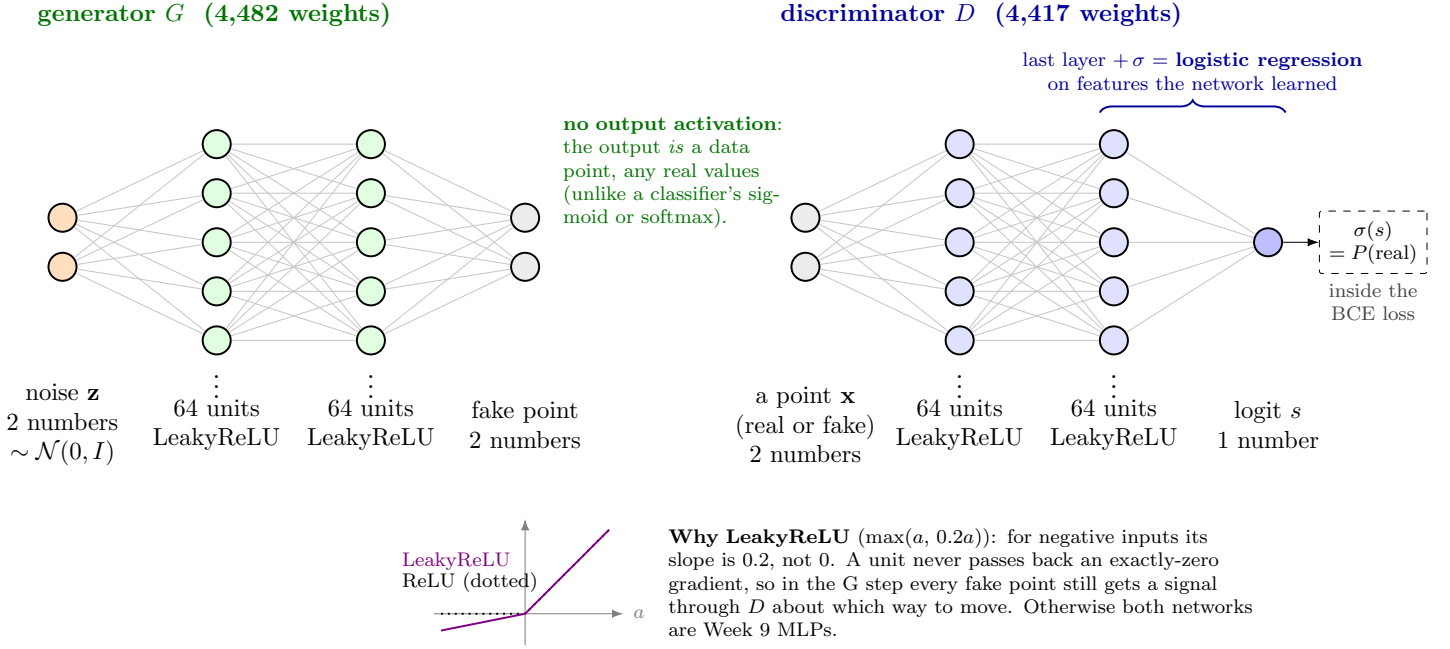


Figure 1: The two networks (HW5 sizes; five of each 64-unit layer drawn). Both are Week 9 MLPs, with three details worth noticing: G ’s output layer has no activation, because it outputs a data point; D ’s last layer followed by a sigmoid is exactly a logistic regression, applied to features the earlier layers learned; and both use LeakyReLU, which keeps gradients flowing for negative inputs.

The discriminator $D_\psi(\mathbf{x}) \in (0, 1)$ is a binary classifier: “probability that $\mathbf{x}$ is real”. Label real samples 1 and generated samples 0, and train with the loss you know from logistic regression (Weeks 3–4), the binary cross-entropy:

$$D\text{'s loss} = -\mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} [\log D(\mathbf{x})] - \mathbb{E}_{\mathbf{z}} [\log (1 - D(G(\mathbf{z})))] . \quad (1)$$

Architecturally, D is HW5 Problem 2’s classifier with one output. Its last layer is usually left as a raw score s (a logit), with $D = \sigma(s)$ applied inside the loss (`BCEWithLogitsLoss`) for numerical stability.

The GAN paper writes the same thing with the opposite sign, as a **value** that D maximizes and G minimizes:

$$V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}} [\log (1 - D(G(\mathbf{z})))] = - (D\text{'s loss}), \quad \min_G \max_D V(D, G). \quad (2)$$

3.1 The best possible detective

Fix the generator, so p_g is fixed, and ask which D maximizes V . Write both expectations as integrals over $\mathbf{x}$:

$$V = \int \left[p_{\text{data}}(\mathbf{x}) \log D(\mathbf{x}) + p_g(\mathbf{x}) \log (1 - D(\mathbf{x})) \right] d\mathbf{x}.$$

Each $\mathbf{x}$ can be optimized separately. With $a = p_{\text{data}}(\mathbf{x})$, $b = p_g(\mathbf{x})$, maximize $a \log D + b \log(1 - D)$ over $D \in (0, 1)$. Setting the derivative $\frac{a}{D} - \frac{b}{1-D}$ to zero gives

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

It is the Bayes posterior “real” vs. “fake” with equal priors, just as in HW4’s generative classifiers.

Worked example: a three-point world Outcomes $x \in \{1, 2, 3\}$ with $p_{\text{data}} = (0.5, 0.5, 0)$ and $p_g = (0.25, 0.25, 0.5)$. Then

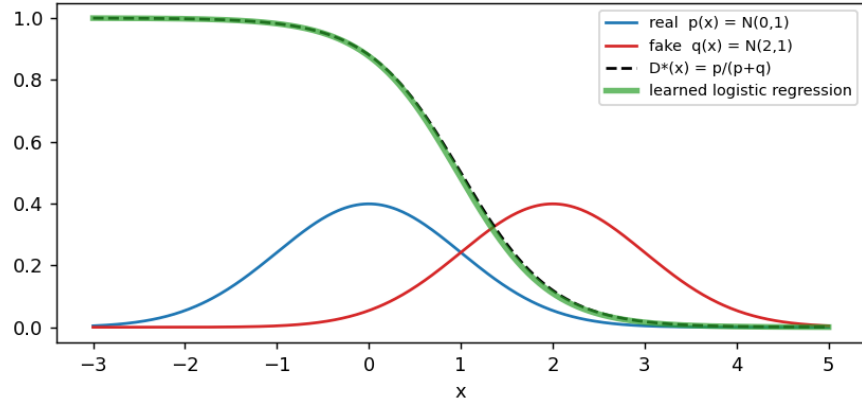
$$D^* = \left(\frac{0.5}{0.75}, \frac{0.5}{0.75}, \frac{0}{0.5} \right) = \left(\frac{2}{3}, \frac{2}{3}, 0 \right).$$

The detective is sure that $x = 3$ is fake (real data never lands there) and leans “real” on $x = 1, 2$. The value is $V = 0.5 \ln \frac{2}{3} \cdot 2 + 0.25 \ln \frac{1}{3} \cdot 2 + 0.5 \ln 1 = -0.405 - 0.549 = -0.955$.

Worked example: two Gaussians: the detective is exactly a logistic regression Real $\mathcal{N}(0, 1)$, fake $\mathcal{N}(2, 1)$. Then

$$D^*(x) = \frac{e^{-x^2/2}}{e^{-x^2/2} + e^{-(x-2)^2/2}} = \frac{1}{1 + e^{2x-2}} = \sigma(2 - 2x),$$

a sigmoid of a *linear* function of x : logistic regression with weight -2 and bias 2 . This is the equal-variance Gaussian (LDA) result from HW4. Values: $D^*(0) = 0.881$, $D^*(1) = 0.5$ (halfway between the two means), $D^*(2) = 0.119$. A logistic regression trained with BCE on samples finds $w = -2.03$, $b = 1.96$ (demo):



4 What the Generator Is Really Minimizing

This section only reuses a tool: the KL divergence from the VAE lecture, applied twice. It explains *why* the game works; the training loop itself does not need it.

Plug the best detective (Eq. 3) back into V and write $m = \frac{1}{2}(p_{\text{data}} + p_g)$, so that $p_{\text{data}} + p_g = 2m$:

$$\begin{aligned} V(D^*, G) &= \int p_{\text{data}} \log \frac{p_{\text{data}}}{p_{\text{data}} + p_g} + \int p_g \log \frac{p_g}{p_{\text{data}} + p_g} = \int p_{\text{data}} \log \frac{p_{\text{data}}}{2m} + \int p_g \log \frac{p_g}{2m} \\ &= -2 \log 2 + \text{KL}(p_{\text{data}} \parallel m) + \text{KL}(p_g \parallel m). \end{aligned}$$

That is,

$$V(D^*, G) = -\log 4 + 2 \text{JSD}(p_{\text{data}} \parallel p_g), \quad \text{JSD}(p \parallel q) = \frac{1}{2} \text{KL}(p \parallel m) + \frac{1}{2} \text{KL}(q \parallel m), \quad m = \frac{1}{2}(p + q). \quad (4)$$

The **Jensen–Shannon divergence** compares each distribution with their average. It is built from two KLs (VAE note), it is symmetric and finite, and it is 0 exactly when $p = q$.

Worked example: JSD on the three-point world $m = (0.375, 0.375, 0.25)$. $\text{KL}(p_{\text{data}} \parallel m) = \ln \frac{4}{3} = 0.288$ and $\text{KL}(p_g \parallel m) = 0.5 \ln \frac{2}{3} + 0.5 \ln 2 = 0.144$, so $\text{JSD} = \frac{1}{2}(0.288 + 0.144) = 0.216$. Check: $-\log 4 + 2(0.216) = -1.386 + 0.432 = -0.955 = V \checkmark$.

So, against a perfect detective, the generator is minimizing a genuine divergence between its distribution and the data's, even though it never computes either density. The minimum is at $p_g = p_{\text{data}}$. There $D^* \equiv \frac{1}{2}$ everywhere (the detective can only guess), and

$$V = -\log 4, \quad \text{i.e. } D\text{'s loss (Eq. 1)} = \log 4 = 2 \ln 2 \approx 1.386$$

($\ln 2 \approx 0.693$ from the real half and 0.693 from the fake half). A GAN whose D -loss hovers near 1.386 has a discriminator that is doing no better than a coin flip.

Key point: Compare the three objectives. EM and the VAE minimize (a bound on) $\text{KL}(p_{\text{data}} \parallel p_{\text{model}})$, i.e. maximum likelihood. A GAN with an ideal detective minimizes $\text{JSD}(p_{\text{data}} \parallel p_g)$. Different divergence, same goal: make the model's distribution match the data's.

Remark. In practice D is never optimal, since it is a finite network trained for a few steps. So Eq. (4) describes what the game is aiming at, not a guarantee about any particular run.

5 Training: Two Networks, Alternating Steps

This is where the GAN is genuinely new: the first two NEW rows of Table 1. This section fills in the details of the loop in §1. Figure 2 draws it as three panels: the two alternating steps, and what is left after training. Then: what exactly flows backward in each step, and why the fakes are detached.

Backprop view (Week 9). In the G step, $D \circ G$ is simply a deeper network: noise $\rightarrow G$'s layers $\rightarrow D$'s layers $\rightarrow$ loss. The chain rule passes $\partial \text{loss} / \partial \mathbf{x}$ from D 's input back into G 's output layer. That vector is literally “which way should this fake point move to look more real”. The G step then updates only G 's weights.

Why detach in the D step? Without it, the D step's backward pass would also compute gradients for G , pointing the *wrong* way (toward making fakes easier to catch). At best this is wasted work; at worst it corrupts G if those gradients are ever applied.

6 The Non-Saturating Generator Loss

The third NEW row of Table 1: a one-line trick, and the reason for it.

The minimax form says G should *minimize* $\log(1 - D(G(\mathbf{z})))$. Write $D = \sigma(s)$ with s the logit that D assigns to a fake. Two candidate G losses:

$$\underbrace{\log(1 - \sigma(s))}_{\text{saturating (original)}} : \left| \frac{\partial}{\partial s} \right| = \sigma(s) = D, \quad \underbrace{-\log \sigma(s)}_{\text{non-saturating}} : \left| \frac{\partial}{\partial s} \right| = 1 - \sigma(s) = 1 - D.$$

Both push s up (make the fake look more real). Early in training G is bad, D catches its fakes easily, and $D(G(\mathbf{z}))$ is near 0:

$D(G(\mathbf{z}))$	0.001	0.01	0.1	0.5
saturating $ \partial/\partial s $	0.001	0.01	0.1	0.5
non-saturating $ \partial/\partial s $	0.999	0.99	0.9	0.5
ratio	999×	99×	9×	1×

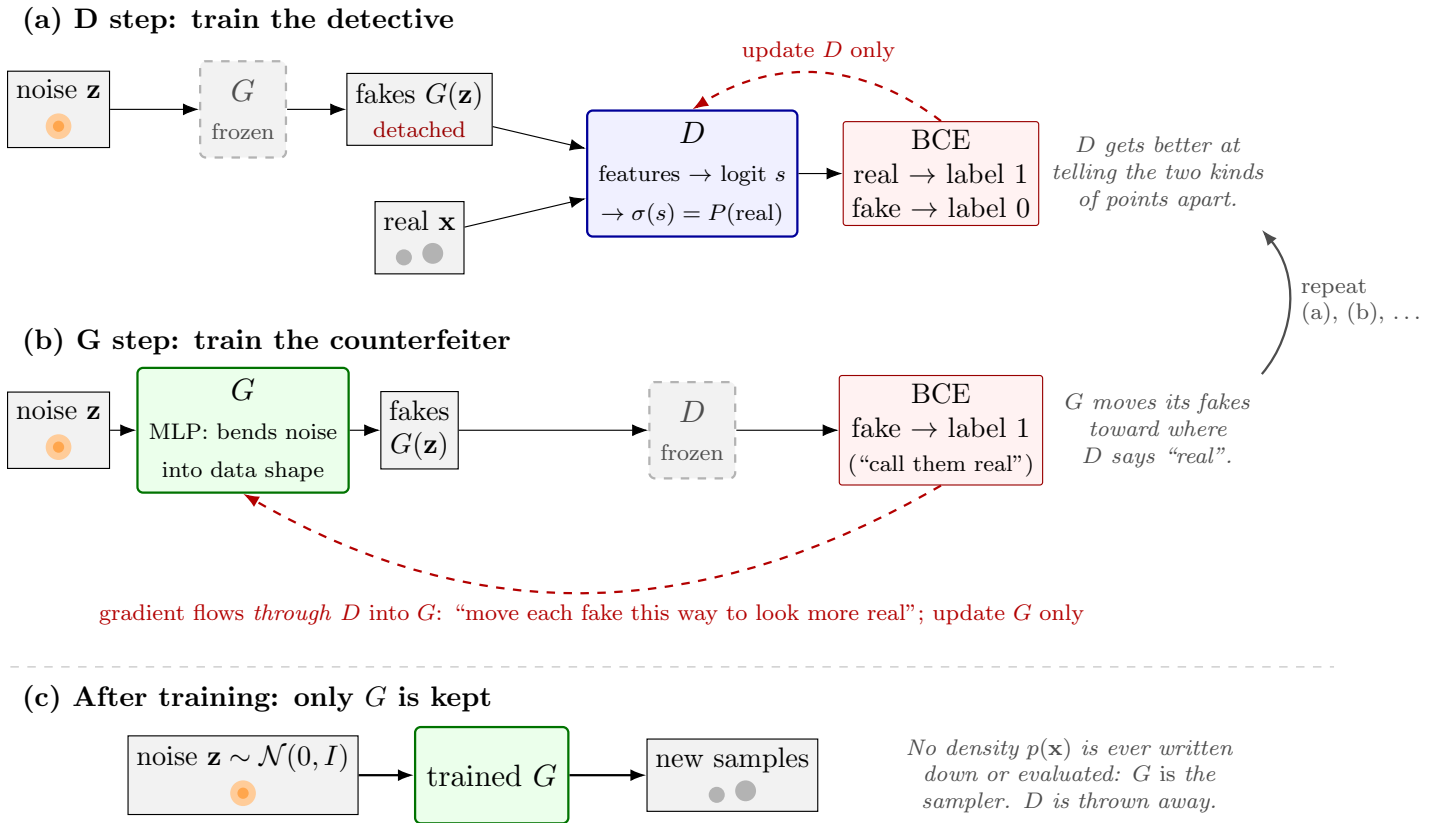
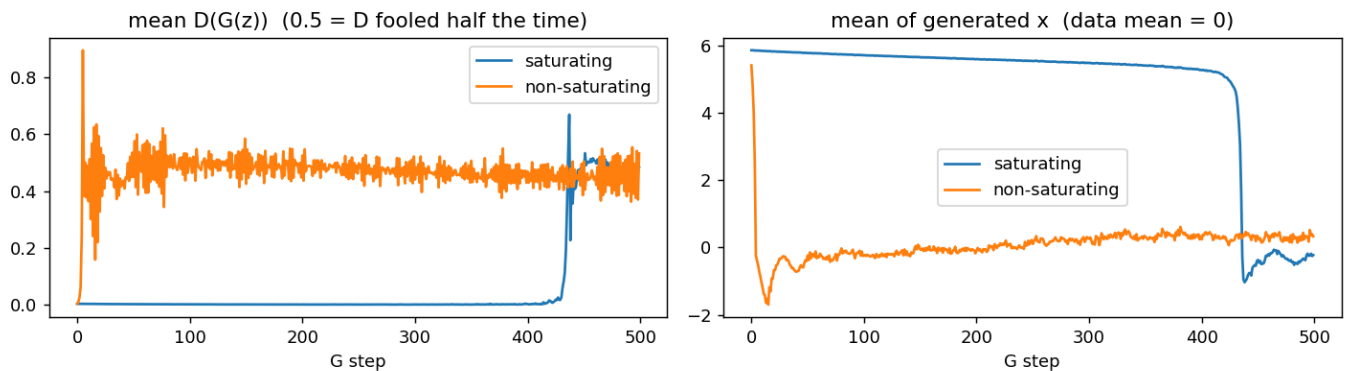


Figure 2: The GAN at a glance. In each step one network learns and the other is *frozen* (dashed gray). **(a)** D learns to label real points 1 and fakes 0; the fakes are detached, so nothing flows back into G . **(b)** G learns to make D output “real” for its fakes; the gradient passes through D ’s layers (without changing them) into G . **(c)** After training, D is discarded and G alone generates.

The original loss gives the *weakest* signal exactly when the generator most needs to move. The non-saturating loss is simply “BCE with label 1 on the fakes”. It is the logistic-regression loss for the *flipped* label, and it is what practically every implementation trains, including HW5’s.



Demo, with plain SGD: G starts far from the data and D gets a head start. At the first step the generator’s gradient norm is 0.019 (saturating) vs. 4.60 (non-saturating). The saturating run stays stuck for about 430 steps; the non-saturating one reaches $D(G(\mathbf{z})) \approx 0.5$ within a few dozen steps.

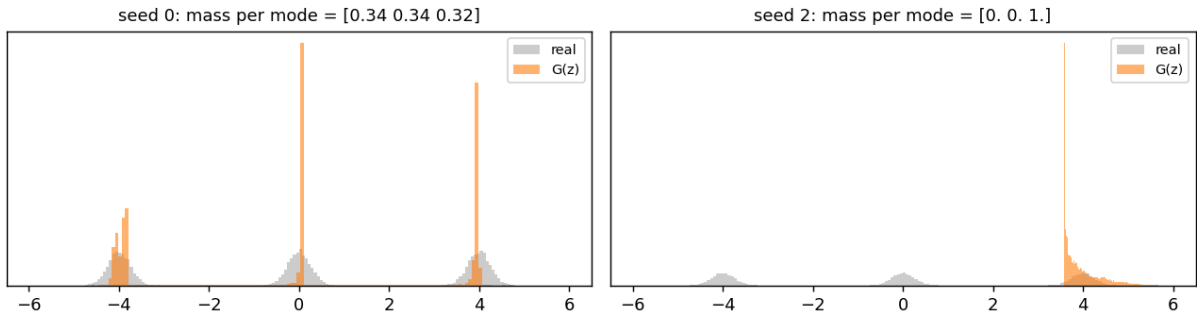
Remark. With the Adam optimizer the difference is much smaller, because Adam rescales each gradient by its own running magnitude (Week 9 / NN-ADVANCED). Practical GANs use both: Adam *and* the non-saturating loss.

7 Common Failure Modes

- **Mode collapse.** Nothing in the game rewards *diversity*. G is only rewarded for fooling the *current* D .

Worked example: collapse on two points Real data: half at $x = -2$, half at $x = +2$. Suppose G outputs $+2$ every time. The best detective is $D^*(-2) = \frac{0.5}{0.5+0} = 1$ and $D^*(+2) = \frac{0.5}{0.5+1} = \frac{1}{3}$. The detective is suspicious of $+2$ (too many samples there) and certain about -2 . But G 's gradient only tells each fake how to move *locally*, and a single network maps nearby noise to nearby outputs. So G tends to move its whole cluster at once, possibly to -2 . Then D adapts, suspects -2 , and the cluster moves back. The pair can chase each other instead of settling on “half here, half there”.

The demo runs the same code on three bumps with two seeds. One run covers all three modes; the other puts all its mass on one:

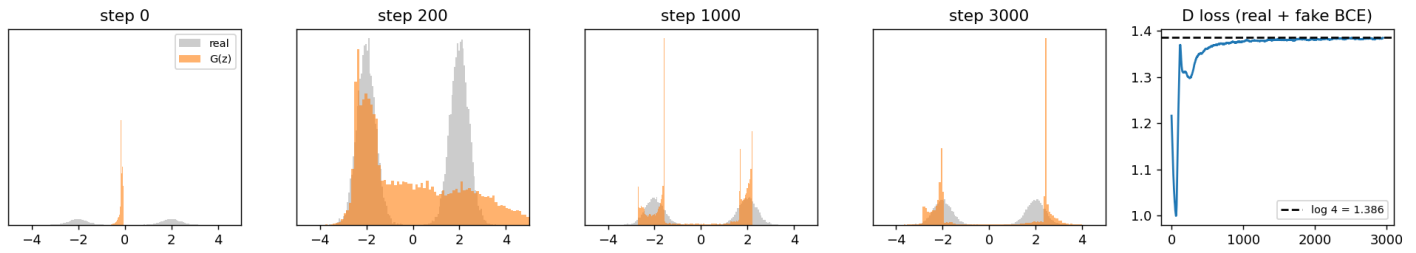


Contrast with a likelihood fit such as EM. Every real point contributes $\log p(\mathbf{x})$ to the objective, so leaving a whole mode of real points poorly explained makes those terms very negative, and ignoring a mode is expensive. The GAN game has no such per-point penalty for data the generator never visits.

- **Vanishing generator gradients.** If D becomes near-perfect early, $D(G(\mathbf{z})) \approx 0$ everywhere and its output is flat around the fakes, so even the non-saturating loss gets little direction. Keep D small, and do not give it many extra steps. (This matters most with the saturating loss. With the non-saturating loss and a small D , a few extra D steps usually help; see the next item.)
- **Who moves faster: the balance between D and G .** Each G step follows D 's *current* opinion of where fakes look real. If G takes several steps for every D step, it keeps chasing an opinion that is out of date: it finds a region D currently likes, piles all its points there, and nothing corrects it until D finally catches up. That is how a faster G turns mild collapse into total collapse. If instead D takes a few steps per G step, then whenever G moves, D 's opinion already reflects *where the fakes are now*, against the whole real distribution, so an empty or over-crowded mode is noticed and corrected sooner. (The limit is the previous item: a D that becomes near-perfect gives flat gradients.) A useful way to read a GAN's output is to check two things per mode: its *share* of the samples (did G put the right amount of mass there?) and its *spread* (is that mass as wide as the real mode?). A partial collapse can get the shares right while squeezing one mode too narrow, the 2-D version of the “spikier fakes” in the 1-D demo (§8).
- **Losses that do not go down.** Training is a two-player game, not a minimization. D 's and G 's losses oscillate, and a flat D -loss near $\log 4$ is what *success* looks like (§4). Judge a GAN by its

samples (plots), not by a loss curve going to zero. GANs are the only models in this course where “the loss isn’t decreasing” is not, by itself, a bug.

8 Demo: A Full 1-D GAN



Real data: two bumps at ± 2 . G and D are small MLPs trained with Adam and the non-saturating loss. At step 0 the fakes are a narrow blob; by step 200 they have spread; by 1000–3000 both modes are covered, with half the mass on each (fraction below 0: 0.50). D ’s loss settles at $1.383 \approx \log 4$. The fakes are spikier than the real bumps: D is a small network and cannot “see” fine differences in shape, so G is not pushed to fix them. HW5 Problem 4 is the same procedure in 2-D.

9 Three Ways to Build a Generative Model

HW4 Problem 3 (EM) and HW5 Problem 4 (GAN) fit the same toy 2-D data, `2gaussian.txt`. HW5 Problem 3’s VAE optimizes the same kind of evidence lower bound as EM, but on images.

	GMM / EM (HW4)	VAE (HW5 P3)	GAN (HW5 P4)
density $p(\mathbf{x})$	explicit, exact	explicit, lower-bounded (ELBO)	never computed (implicit)
latent / noise	discrete component k	continuous code, inferred by an encoder	noise only, no encoder
training signal	likelihood	ELBO = reconstruction – KL	a discriminator’s verdicts
divergence (ideal)	$\text{KL}(p_{\text{data}} \parallel p)$	$\text{KL}(p_{\text{data}} \parallel p)$ (via a bound)	$\text{JSD}(p_{\text{data}} \parallel p_g)$
dynamics	monotone (EM never decreases)	roughly monotone (one loss)	a game: oscillates
typical failure	local optima	blurry samples, posterior collapse	mode collapse, instability

10 HW5 Problem 4: Map and Implementation Notes

- HW5 Problem 4’s code is *complete*: you run it, read it, and explain what you see. The questions map onto this note as follows.
- **(A)(i) the training step**: §5 (why the D step detaches the fakes) and §6 (why the G step uses label 1: the non-saturating loss). Networks: small MLPs, no activation on G ’s output, raw logit out of D ; the data are standardized so that G ’s outputs start on the right scale.

- **(A)(ii) the losses:** §4 (why D's loss sits at $2 \ln 2 = \log 4$ when it can't tell real from fake). Early fakes sit near where G 's initial outputs are; by the end they should cover both modes of `2gaussian.txt`.
- **(A)(iii) implicit density:** §2 (why p_g has no formula) and §4 (what the game optimizes instead).
- **(B) three modes:** §7 (mode collapse, and why EM is far less prone to it). Compare the printed shares *and* spreads: a mode can get the right number of points but the wrong shape.
- **Optional Problem 7** (DCGAN): the same training step; only G and D change.

Take-home points

- To generate you need a sampler, not a density. A GAN learns only the sampler $x = G(z)$.
- The discriminator is a (deep) logistic regression. The best one is $D^* = p_{\text{data}} / (p_{\text{data}} + p_g)$, and against it the generator minimizes the Jensen-Shannon divergence. At equilibrium $D \equiv \frac{1}{2}$ and D's loss is $\log 4$.
- Train by alternating a D step (fakes detached) and a G step (gradient through D). Use the non-saturating loss.
- Expect oscillation and watch for mode collapse. Judge a GAN by its samples.

References

- Goodfellow et al. (2014), *Generative Adversarial Nets*, NeurIPS.
- Goodfellow (2016), *NIPS 2016 Tutorial: Generative Adversarial Networks*, arXiv:1701.00160.
- d2l book, §20.1 (GANs) and §20.2 (DCGAN).