▶ Notebook

Lecture 2 at a glance: six parts, twenty-seven sections

1 · Why generate?
1.1 Things that do not exist
1.2 Text, pictures and sound
1.3 Beyond pictures
1.4 What they all share

2 · Fit a distribution, then sample
2.1 The data: Old Faithful
2.2 One Gaussian: fit, score, sample
2.3 Two bumps: a hidden choice
2.4 Fit the mixture, then sample it
2.5 How many bumps?

3 · Images: PCA and Gaussians
3.1 An image is 784 numbers
3.2 A bell for every pixel
3.3 A few numbers per digit
3.4 Generate: sample a code, decode it

4 · Autoencoders
4.1 From PCA to an autoencoder
4.2 Train one
4.3 How many numbers in the code?
4.4 Where the codes land

5 · Can an autoencoder generate?
5.1 Decode a random code
5.2 Where the codes are
5.3 What we want instead

6 · The variational autoencoder
6.1 Two roads to the VAE
6.2 Road 1: from the autoencoder
6.3 Road 2: from a generative model
6.4 The loss, term by term
6.5 What the VAE learned
6.6 The KL weight β
6.7 Wrap-up

Click a section to jump to it. We start with why generative models matter, and end by training a VAE.

▶ Notebook 1.1

Who are these people?



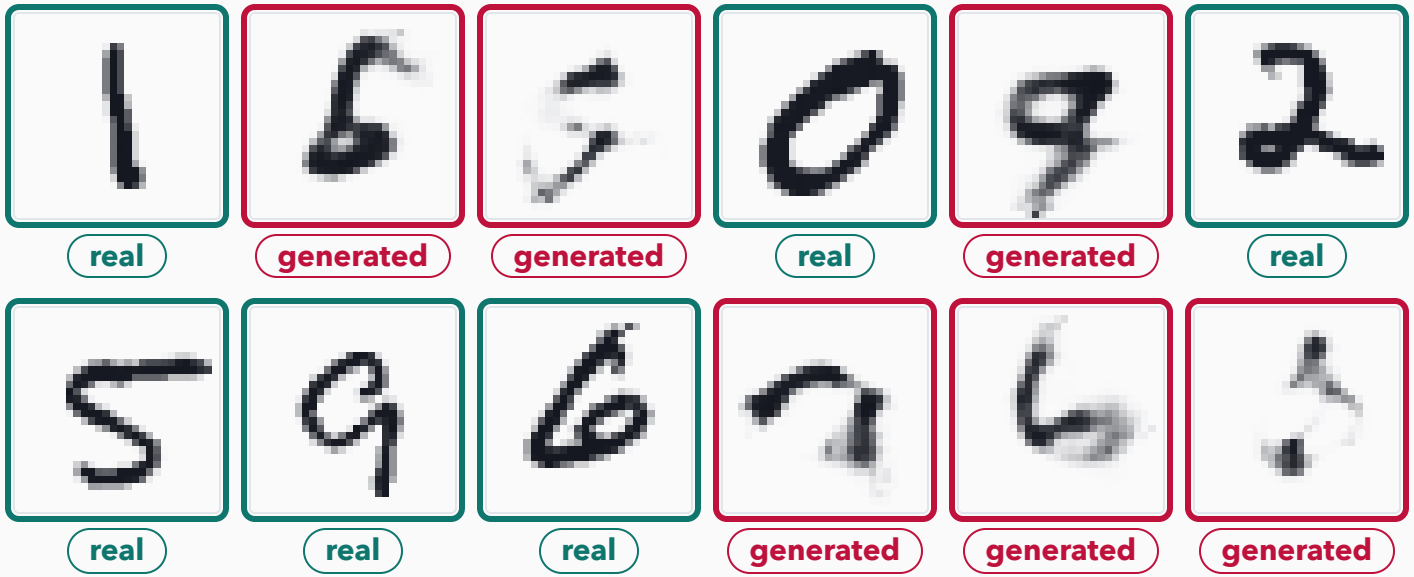
None of them exist. A network turned random numbers into each face.

> Nobody. Each face is a sample from a generative model (StyleGAN, 2019): random numbers in, a face out.

Faces generated by StyleGAN: Karras, Laine and Aila, "A Style-Based Generator Architecture for GANs", CVPR 2019. NVIDIA, CC BY-NC 4.0.

▶ Notebook 1.1

Real or generated? Six of these twelve digits are fake



Six are handwritten MNIST test digits. Six come from a network.

The fakes: numbers 2, 3, 5, 10, 11, 12 (counting left to right, top to bottom).

By the end of today you will have trained the network that made them: a VAE with 16 random numbers per digit.

> Answer: the six outlined in red and labelled "generated". Some are easy (broken strokes); some fool most people.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 1.2

A chatbot is a generative model: one prompt, three different answers

Prompt: Write one sentence that opens a short story about the first monsoon rain in Gandhinagar.

Sample 1. As the first heavy drops of the monsoon struck the sunbaked avenues of Sector 21, the parched earth of Gandhinagar exhaled a sweet, dusty petrichor that signaled the end of the long summer.

Sample 2. As the first heavy drops struck the sun-baked asphalt of Sector 17, Gandhinagar exhaled its scorching summer heat in a sweet, petrichor-laced sigh that rippled down the quiet, tree-lined avenues.

Sample 3. The brutal summer heat surrendered in a single afternoon when heavy, bruised clouds cracked open over Gandhinagar, washing its dusty neem trees and quiet, grid-lined sectors in the intoxicating scent of the season's first rain.

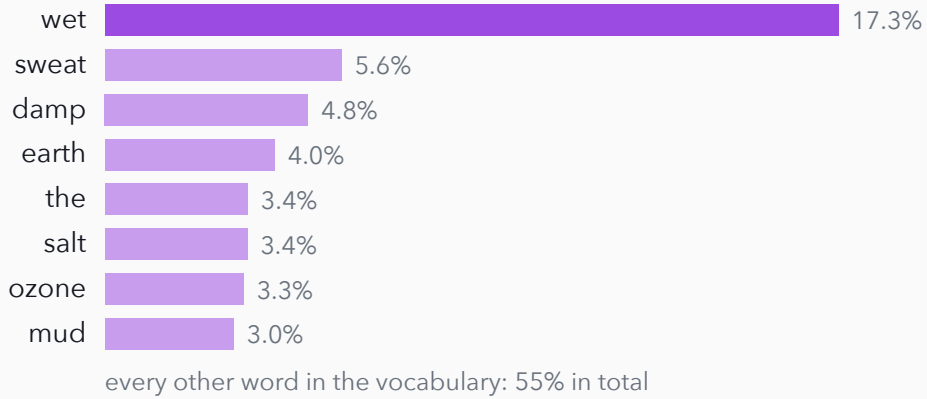
Same prompt, three different replies. Each is a sample from a distribution over sentences.

Generated for this lecture on 21 October 2026 with Gemini 3.5 Flash, temperature 1 (`assets/examples/provenance.json`).

▶ Notebook 1.2

Under the hood: a probability for every possible next word

"After months of heat, the first monsoon rain smelled of …"



Sampling = pick a word with these probabilities, append it, repeat.

The same model's three continuations: "melons and oranges.", "sweat and damp ruins. It was moist, it was s", "desperation. What had occurred, and what transpired that its".

> Sample a word, append it, repeat. That is all a chatbot does when it writes, at a much larger scale.

SmolLM2-360M-Instruct (Hugging Face), next-token probabilities and three samples at temperature 1, computed for this lecture.

▶ Notebook 1.2

Text to image: one prompt, four samples

Prompt: A watercolour painting of a camel walking across the white salt desert of the Rann of Kutch at sunrise, soft light, wide view.



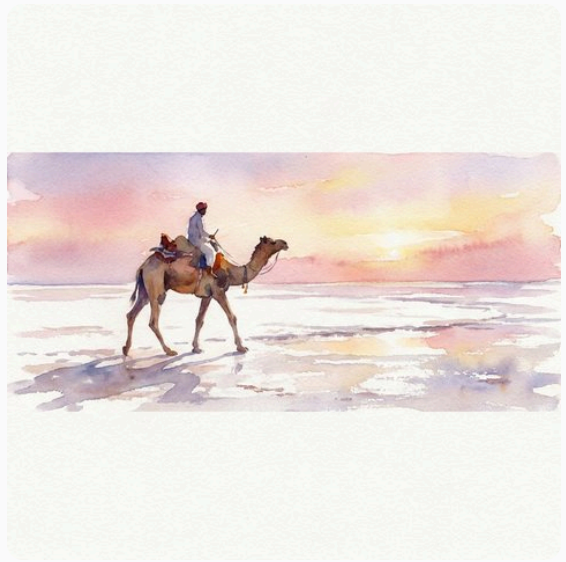
sample 1



sample 2



sample 3



sample 4

Three more runs, same prompt: a rider appears, a walker, a different horizon. Four samples.

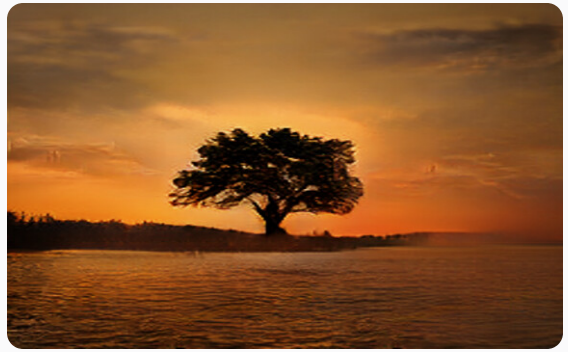
Generated for this lecture on 21 October 2026 with Gemini 3 Pro Image (`assets/examples/provenance.json`).

▶ Notebook 1.2

Image in, image out


one label map


sample 1


sample 2


sample 3

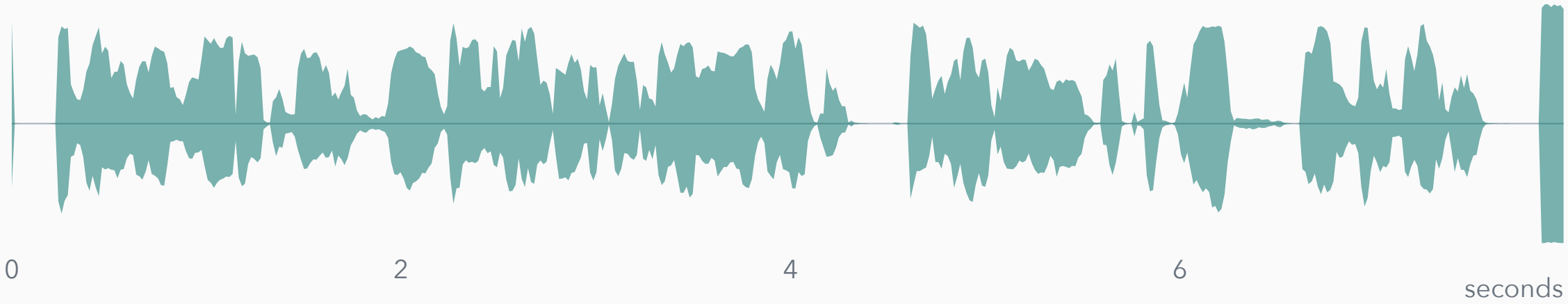
> The same label map, sampled three times: three different photographs. Still a distribution, now given an input.

Isola et al., pix2pix (CVPR 2017); Park et al., SPADE (CVPR 2019); Zhu et al., CycleGAN (ICCV 2017). Attributed teaching excerpts; copyright with the authors.

▶ Notebook 1.2

Sound: text in, 24,000 numbers per second out

Text: “A generative model turns a few random numbers into a new example: a sentence, a picture, or this voice.”



0:00 / 0:00

191,219 numbers for 8.0 seconds. A few of them: 0.102, 0.089, 0.083, 0.072, 0.062, 0.051, …

The output is just a long list of numbers: the air pressure, 24,000 times a second. Generating sound = generating that list.

▶ Notebook 1.3

Beyond pictures: generative models in science

Weather

GenCast samples 50 possible futures for the next 15 days. The spread of the samples says how sure the forecast is.

Price et al., Nature 2025

Proteins

RFdiffusion generates new protein shapes, for example ones that bind a chosen target.

Watson et al., Nature 2023

Materials

MatterGen proposes new crystal structures with properties you ask for.

Zeni et al., Nature 2025

Missing data

Sensors drop readings. A model of p(x) fills the gap with plausible values, not a flat average, and says how unsure it is.

for example, air-quality and smart-meter records

> Materials, and the everyday case: filling in missing sensor readings with plausible values.

Price et al., "Probabilistic weather forecasting with machine learning" (GenCast), Nature 2025; Watson et al., "De novo design of protein structure and function with RFdiffusion", Nature 2023; Zeni et al., "A generative model for inorganic materials design" (MatterGen), Nature 2025.

▶ Notebook 1.3

What a model of p(x) is good for

Create
new text, images, sound, designs

Fill in
missing pixels, missing readings, the rest of a sentence

Simulate
many plausible futures, not one average

Spot the odd one out
a reading with very low p(x) is suspicious: a faulty sensor, a fraud

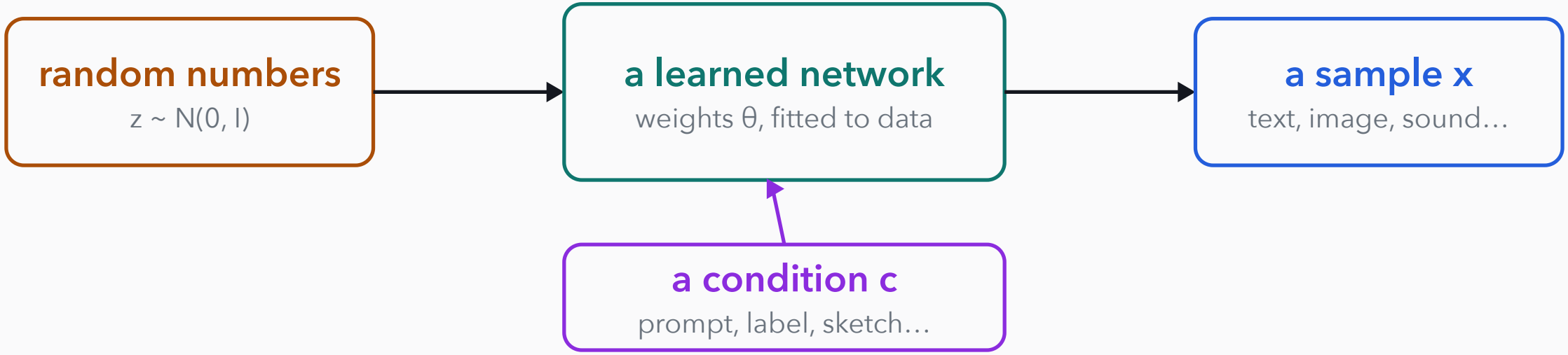
Compress
likely data needs fewer bits: Lecture 1's codes

Make more data
extra training examples, or shareable synthetic records

> Three that need scores: spot anomalies, compress, and generate extra data when real data is scarce or private.

▶ Notebook 1.4

What they share: random numbers in, a learned network, a sample out



> Most also take a condition c that steers the sample: a prompt, a class label, a sketch. Then the model is p(x | c).

▶ Notebook 1.4

Conditions steer the sample; today we learn p(x) without one

	condition c	sample x
chatbot	your prompt	a reply
text to image	a prompt	a picture
pix2pix, SPADE, CycleGAN	a sketch, a map, a photo	a photo
text to speech	a sentence	a waveform
today	none	a 28 × 28 digit

Learning a distribution over images is the hard part, with or without a condition.

> Today we drop the condition and learn p(x) for handwritten digits: the core problem underneath all of these.

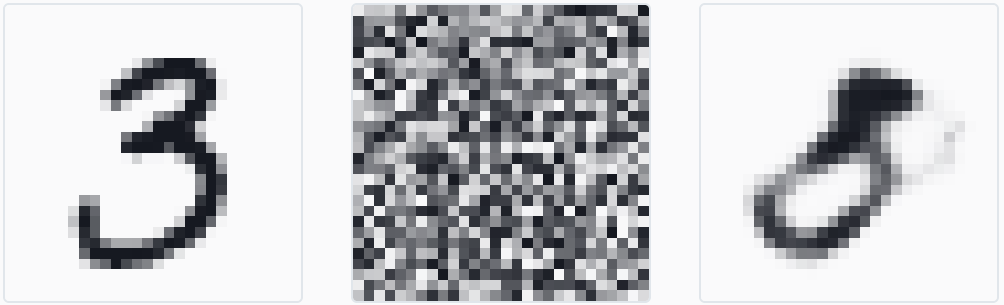
▶ Notebook 1.4

A model of p(x) can do two jobs: score and sample

Score: how plausible is this x?

$$p(\text{a clean 3}) \text{ high}, \qquad p(\text{static}) \text{ tiny}$$

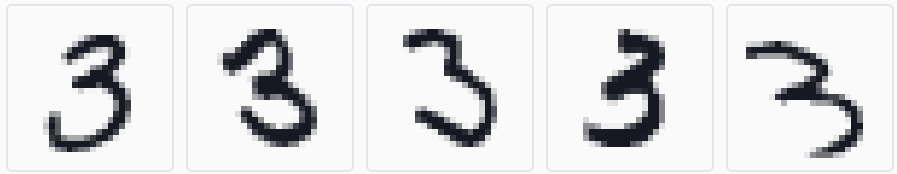
Sample: make a new x that looks like the data.



> Job two, sample: produce a new x. Both jobs come from the same fitted distribution.

▶ Notebook 1.4

Why not just output the average?



average of 100 threes:



Nobody writes this 3. The average is the blur in the middle of many real 3s.

> The average looks like no real 3. We need the whole distribution, so we can draw one sharp 3 at a time.

▶ Notebook 1.4

Today's route: six parts

1 · Why generate | 2 · Fit, then sample | 3 · PCA and Gaussians | 4 · Autoencoders | 5 · Can it generate? | 6 · The VAE

1 · Why generate?
what generative models do, and the shared recipe

2 · Fit, then sample
one number per example: a Gaussian, then two bumps

3 · PCA and Gaussians
784 numbers per digit: compress first, then fit

4 · Autoencoders
a learned, bent version of PCA

5 · Can it generate?
random codes, and why they fail

6 · The VAE
a recipe we can sample, trained with Lecture 1's ELBO

> From one number per example to 784, from fitting by formula to training networks, ending with the variational autoencoder.

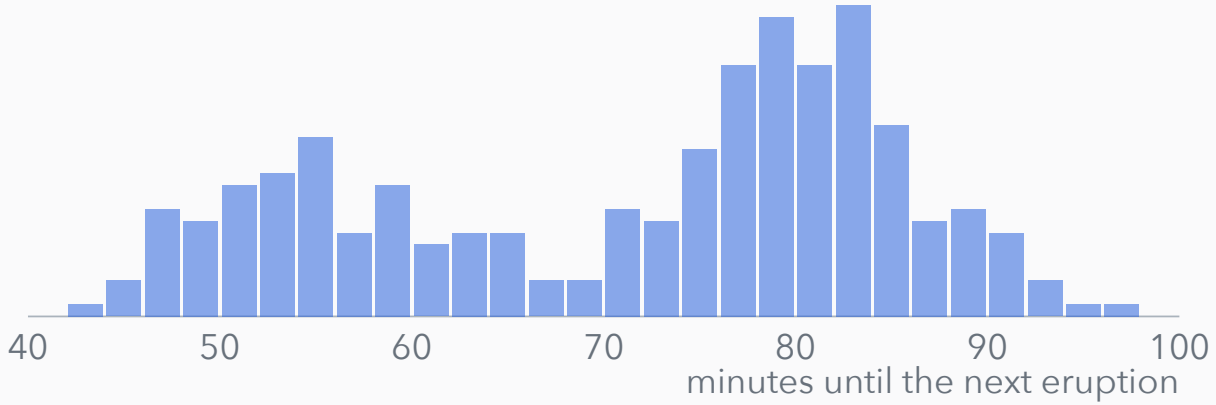
▶ Notebook 2.1

Old Faithful: 272 eruptions, and the wait before each one

A geyser in Yellowstone. Rangers record how long visitors wait between eruptions.

The first five waits, in minutes: **79, 54, 74, 62, 85**, …

All 272 waits as a histogram: two clumps, around 55 and 80 minutes.



> The histogram shows two clumps: short waits around 55 minutes, long ones around 80. Few waits near 70.

Old Faithful geyser data (Azzalini and Bowman, 1990), as distributed with R (`datasets::faithful`): 272 eruptions.

▶ Notebook 2.1

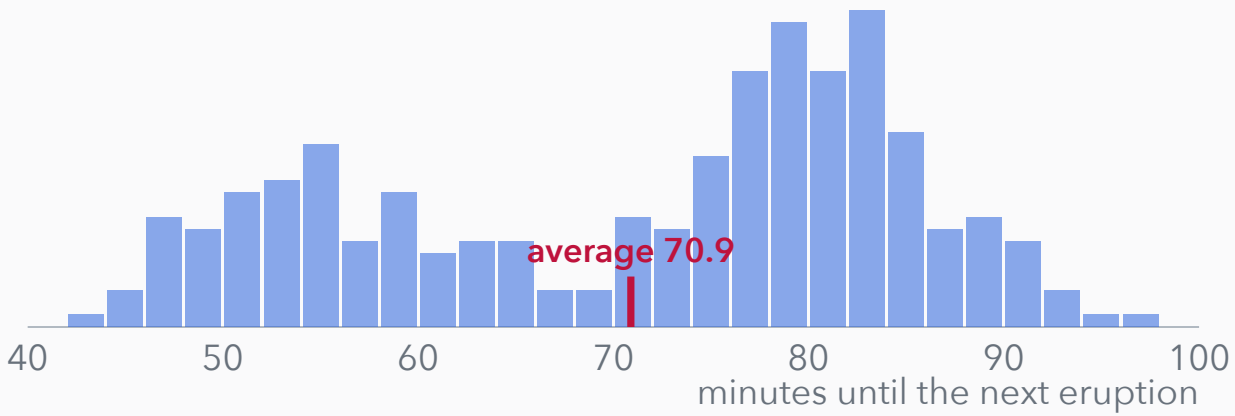
The task: examples in, a distribution out, new examples from it



> Sample: run the fitted distribution forwards to get new waits that nobody measured.

▶ Notebook 2.1

The average wait, 70.9 minutes, is a poor sample



Only **46** of 272 waits lie within 5 minutes of the average.

The average sits in the dip. To make realistic waits we need the shape of the data, not one number.

> A realistic new wait should come from one of the clumps, as often as the clumps occur. That needs a distribution.

Old Faithful geyser data (Azzalini and Bowman, 1990), as distributed with R (`datasets::faithful`): 272 eruptions.

▶ Notebook 2.2

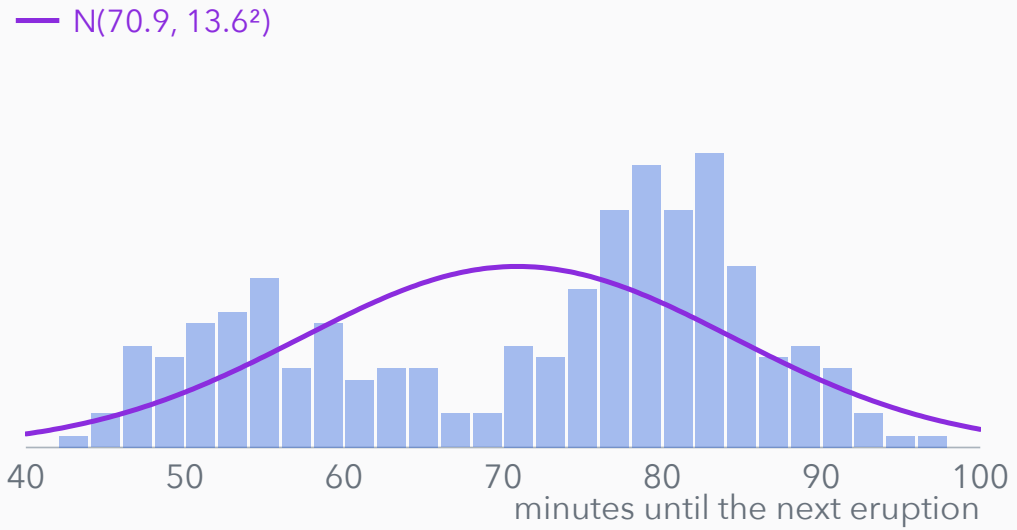
One Gaussian: the best μ and σ have a formula

$$p(x) = \mathcal{N}(x;\, \mu, \sigma^2)$$

$$\text{maximise} \quad \frac{1}{n}\sum_{i=1}^{n} \log \mathcal{N}(x_i;\, \mu, \sigma^2)$$

$$\hat{\mu} = \frac{1}{n}\sum_{i} x_i = 70.9$$

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i} (x_i - \hat{\mu})^2 = 13.6^2$$

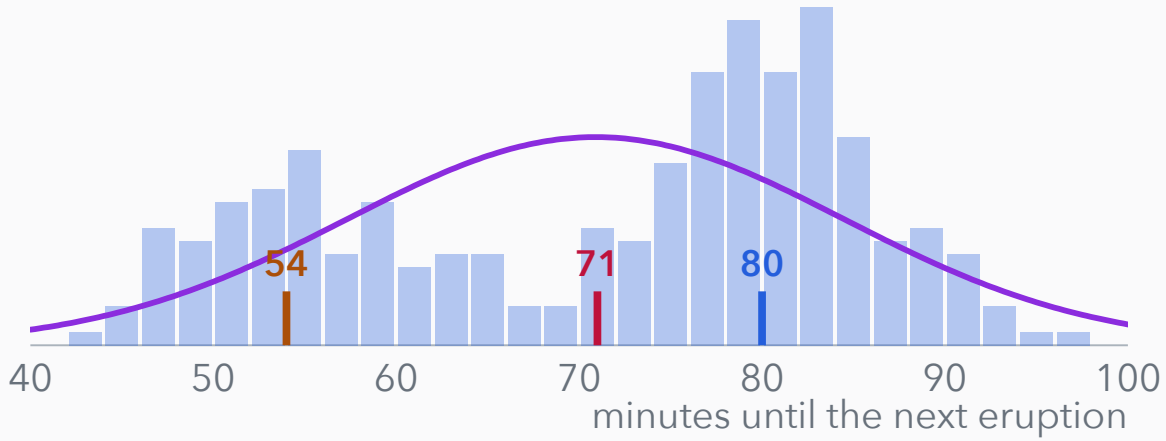


> Set the derivatives to zero: μ̂ is the average and σ̂² the average squared deviation. One bump, centred in the dip.

Old Faithful geyser data (Azzalini and Bowman, 1990), as distributed with R (`datasets::faithful`): 272 eruptions.

▶ Notebook 2.2

Score: how probable is each wait under the bell?



wait x	p(x) under the bell
54 minutes	0.0135
71 minutes	0.0294
80 minutes	0.0235

Average log-likelihood of all 272:
−4.027

> One number scores the whole model: the average log p(x) over the data. We will try to beat it.

Old Faithful geyser data (Azzalini and Bowman, 1990), as distributed with R (`datasets::faithful`): 272 eruptions.

▶ Notebook 2.2

Sample from the bell: the draws fill the empty dip

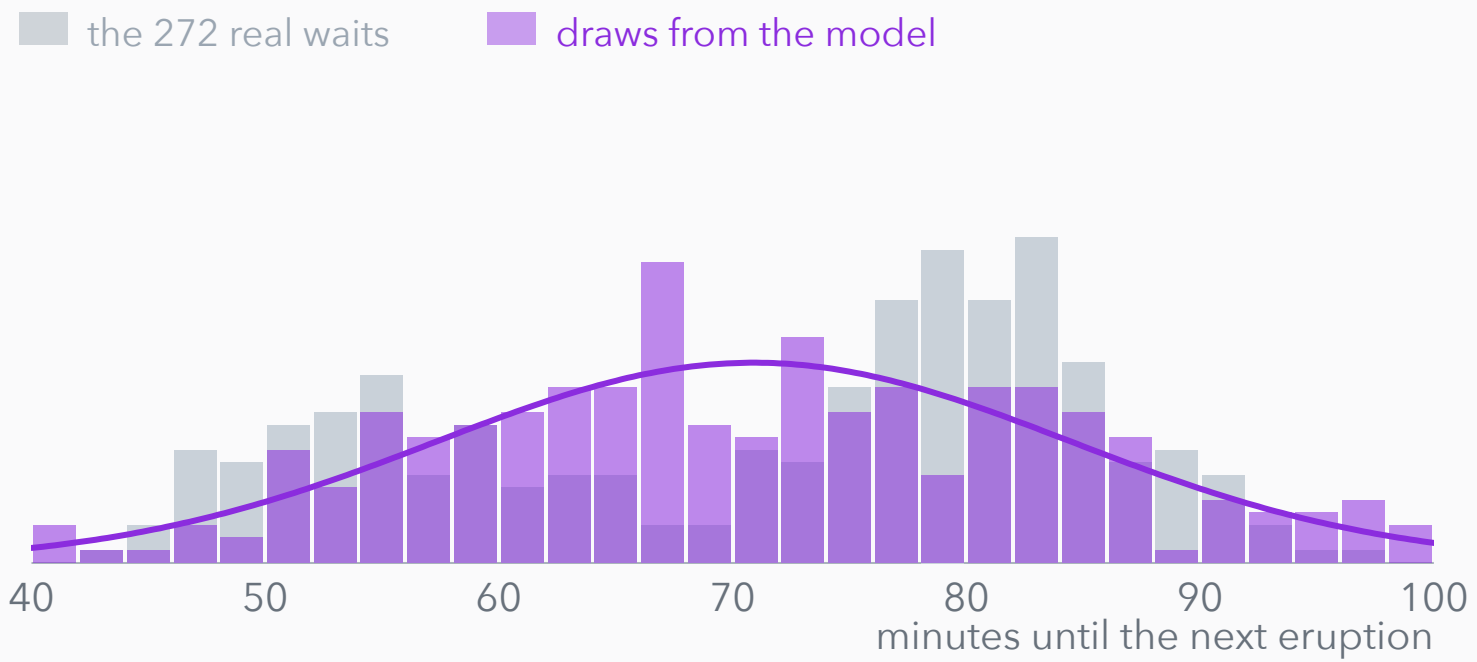
one step: draw a time

$N(70.9, 13.6^2)$
the one fitted Gaussian

latest: 81.4 minutes

73 of 272 draws land in 62–72 min
(the data: 30 of 272)

272 draws so far

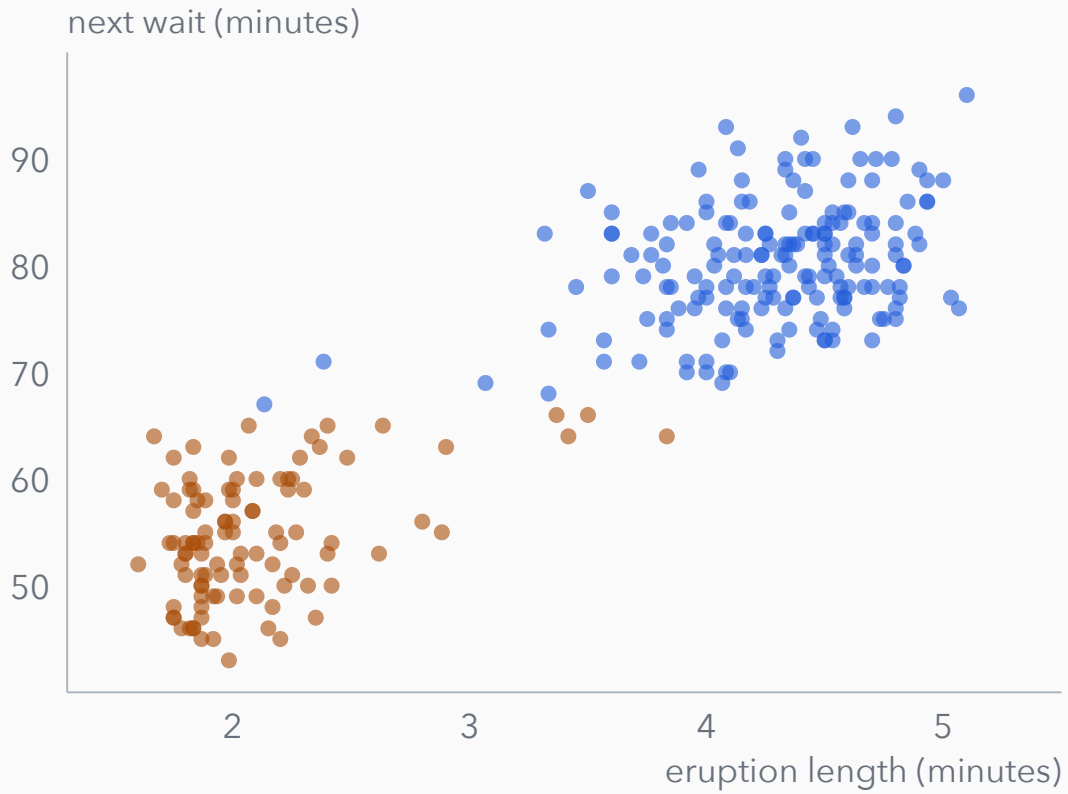


The draws pile up in the dip where real eruptions rarely are, and miss the two clumps.

Old Faithful geyser data (Azzalini and Bowman, 1990), as distributed with R (`datasets::faithful`): 272 eruptions.

▶ Notebook 2.3

The geyser has two kinds of eruption



Add a second measurement: how long each eruption lasted.

Short eruptions (about 2.1 min) come before short waits; long ones (about 4.3 min) before long waits.

The kind of eruption is a hidden choice. If we only see the wait, we never see the kind directly.

> Nature first "chooses" a short or a long eruption; the wait then depends on that choice.

Old Faithful geyser data (Azzalini and Bowman, 1990), as distributed with R (`datasets::faithful`): 272 eruptions.

▶ Notebook 2.3

A recipe with a hidden step: pick a bump, then draw

$$k \sim \text{Categorical}(\pi_1, \pi_2) \qquad \text{(short or long)}$$

$$x \mid k \sim \mathcal{N}(\mu_k, \sigma_k^2) \qquad \text{(the wait, given the kind)}$$

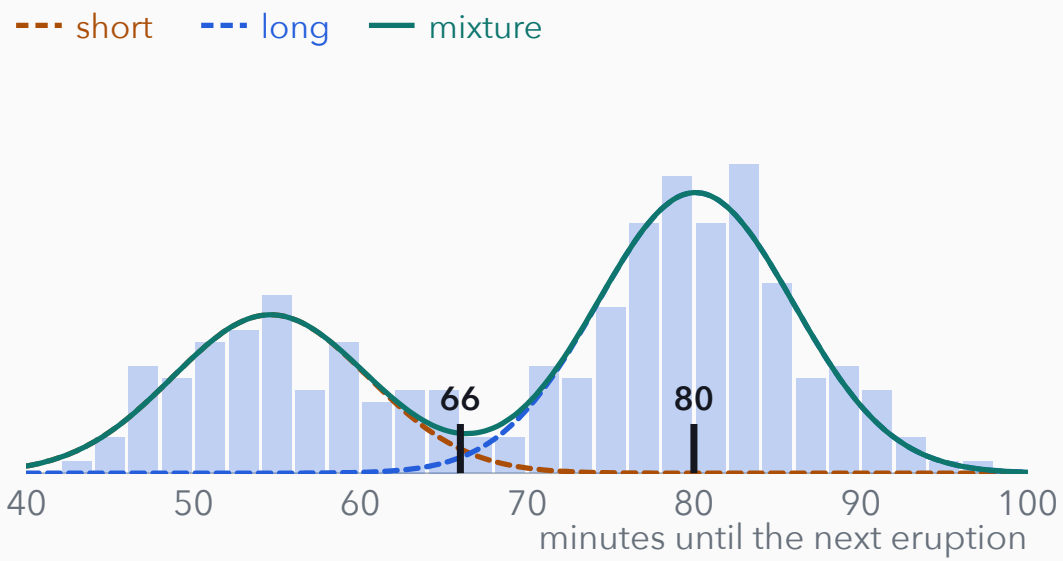
k is a **latent** (hidden) variable: it shapes the data, but we never observe it.

$$p(x) = \sum_{k=1}^{2} \pi_k \, \mathcal{N}(x;\, \mu_k, \sigma_k^2) \qquad \text{a mixture of Gaussians}$$

> The distribution of x alone adds up both ways x could have been made: a mixture of Gaussians.

▶ Notebook 2.3

To score a wait, add over both kinds



$$p(66) = \pi_1 \mathcal{N}(66;\ \text{short}) + \pi_2 \mathcal{N}(66;\ \text{long})$$

$$= 0.0037 + 0.0024 = 0.0062$$

$$p(80) = 0.000002 + 0.0435 = 0.0435$$

Average log-likelihood: **−3.801**, up from −4.027 for one bump.

> Over all 272 waits the mixture scores −3.801 against −4.027: the data are e^61 times more probable under it.

Old Faithful geyser data (Azzalini and Bowman, 1990), as distributed with R (`datasets::faithful`): 272 eruptions.

▶ Notebook 2.4

What we minimise: minus the average log-likelihood

one wait

$$p(x_i) = \pi_1 \mathcal{N}(x_i;\, \mu_1, \sigma_1^2) + \pi_2 \mathcal{N}(x_i;\, \mu_2, \sigma_2^2)$$

all 272 waits: the score, higher is better

$$\text{score} = \frac{1}{272} \sum_{i=1}^{272} \log p(x_i)$$

flip the sign: the loss, lower is better

$$\text{loss} = -\frac{1}{272} \sum_{i=1}^{272} \log p(x_i)$$

> Flip the sign to get something to minimise: the loss. Now we only need the loop.

Old Faithful geyser data (Azzalini and Bowman, 1990), as distributed with R (`datasets::faithful`): 272 eruptions.

▶ Notebook 2.4

The numbers to learn, each kept valid

	must be	we learn	and use	start
centres μ_1, μ_2	any number	`mu`	μ = `mu`	65 and 75 min
widths σ_1, σ_2	positive	`log_s`	$\sigma = \exp($`log_s`$)$	$\exp(2.3) \approx 10$ min
shares π_1, π_2	positive, add to 1	`w`	$\pi = \mathrm{softmax}($`w`$)$	$\mathrm{softmax}(0, 0) = (½, ½)$

$$\mathrm{softmax}(w)_k = \frac{e^{w_k}}{e^{w_1} + e^{w_2}}$$

> The two shares must be positive and add to 1. Learn any two numbers w and pass them through softmax.

Old Faithful geyser data (Azzalini and Bowman, 1990), as distributed with R (`datasets::faithful`): 272 eruptions.

▶ Notebook 2.4

The loop: one line for each piece of the formula


```
def bell(x, m, s):                          # N(x; m, s²), the formula from 2.2
    return torch.exp(-((x - m) / s)**2 / 2) / (s * math.sqrt(2 * math.pi))

opt = torch.optim.Adam([mu, log_s, w], lr=0.1)
for step in range(300):
    pi, sigma = torch.softmax(w, 0), torch.exp(log_s)
    p_short = pi[0] * bell(x, mu[0], sigma[0])     # π₁ N(x; μ₁, σ₁²)
    p_long  = pi[1] * bell(x, mu[1], sigma[1])     # π₂ N(x; μ₂, σ₂²)
    loss = -torch.log(p_short + p_long).mean()     # -(1/272) Σ log p(x)
    opt.zero_grad(); loss.backward(); opt.step()
```

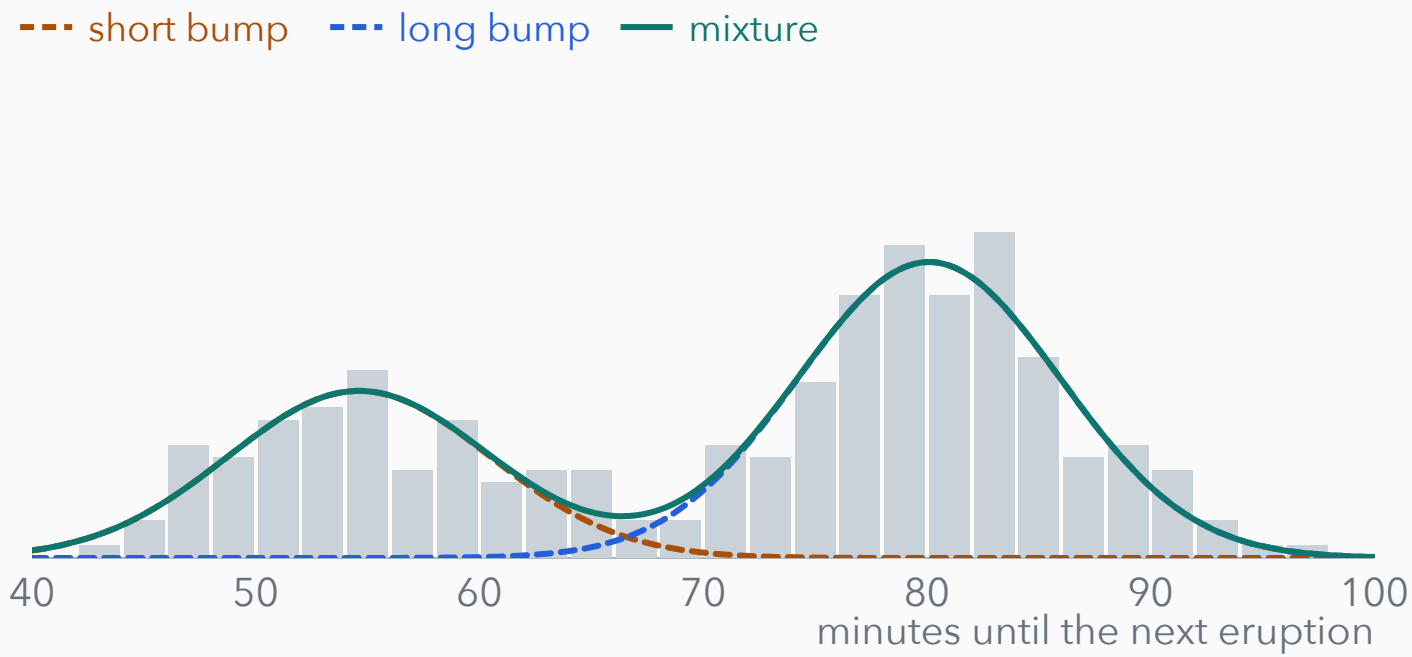
x holds all 272 waits, so `p_short` and `p_long` are 272 numbers each, one per wait.

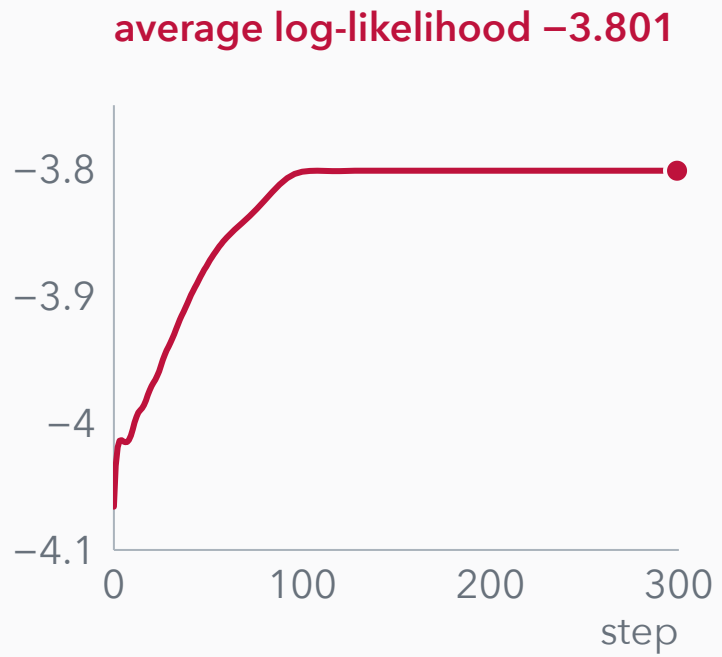
> x is a tensor of all 272 waits, so each line works on every wait at once. backward() finds how the loss changes with mu, log_s and w; step() nudges them.

Old Faithful geyser data (Azzalini and Bowman, 1990), as distributed with R (`datasets::faithful`): 272 eruptions.

▶ Notebook 2.4

Press play: the bumps separate onto the two clumps





They pull apart onto the clumps; the score rises from −4.07 to −3.80.

Old Faithful geyser data (Azzalini and Bowman, 1990), as distributed with R (`datasets::faithful`): 272 eruptions.

▶ Notebook 2.4

It found the two kinds of eruption without being told

	short kind	long kind
share π	0.36	0.64
centre μ (min)	54.6	80.1
width σ (min)	5.9	5.9

Waits it puts in the short bump follow eruptions of **2.09** min on average; long bump: **4.29** min.

It never saw eruption lengths. The hidden variable it invented matches a real one.

> Check against the eruption lengths, which the fit never saw: the two bumps line up with short and long eruptions.

Old Faithful geyser data (Azzalini and Bowman, 1990), as distributed with R (`datasets::faithful`): 272 eruptions.

▶ Notebook 2.4

Sample: pick a bump, then a time

step 1: pick a bump

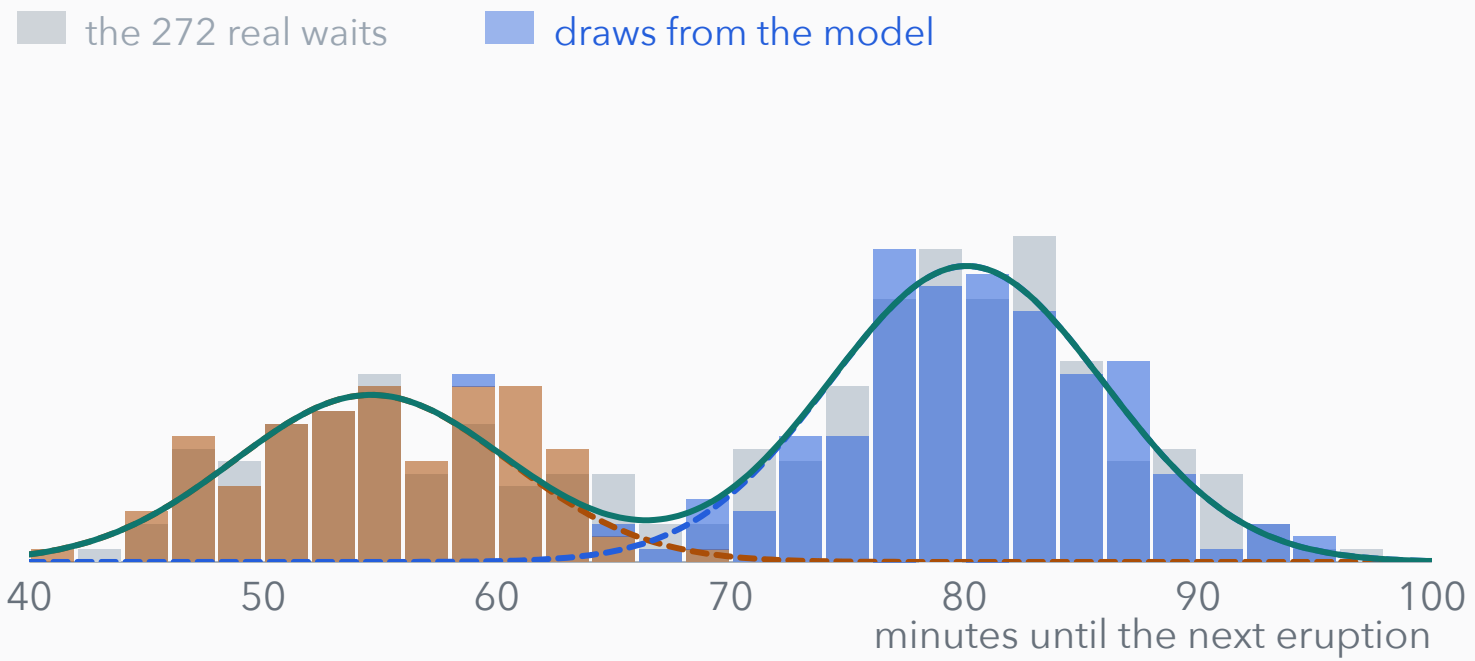
short wait: probability 0.36
N(54.6, 5.9²) · picked 107×

long wait: probability 0.64
N(80.1, 5.9²) · picked 165×

step 2: draw a time from that bump

latest: long bump, 78.9 minutes

272 draws so far

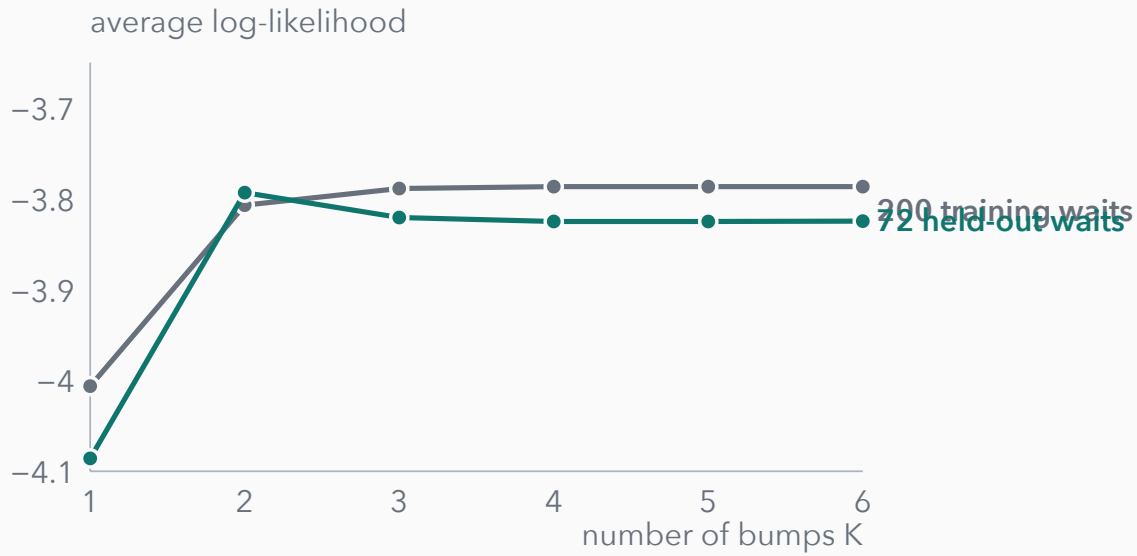


> After 272 draws the histogram of samples matches the data: two clumps, few in the dip.

Old Faithful geyser data (Azzalini and Bowman, 1990), as distributed with R (`datasets::faithful`): 272 eruptions.

▶ Notebook 2.5

More bumps always fit the training waits better; new waits decide



Fit on 200 waits, score on the other 72.

Held-out score is best at **K = 2**, then drops slightly.

> On the 72 held-out waits the score peaks at two bumps. Extra bumps fit noise in the training waits.

Old Faithful geyser data (Azzalini and Bowman, 1990), as distributed with R (`datasets::faithful`): 272 eruptions.

▶ Notebook 2.5

Part 2 in one slide

step	one number per example	what we learned
fit	choose parameters to maximise the average log p(x)	same loop as Lecture 1
score	evaluate log p(x) of any wait	held-out score picks the model
sample	run the recipe: pick a bump, then draw	hidden variables make samples realistic

Next: each example is a 28 × 28 image, 784 numbers. Same three steps, much harder.

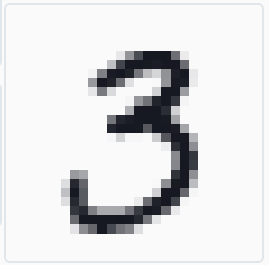
> Next, images: the same three steps with 784 numbers per example.

▶ Notebook 3.1

MNIST: 70,000 handwritten digits, 28 × 28 pixels each



60,000 for training, 10,000 for testing.



Our running example: test digit #90, a 3.

> We follow one digit all lecture: test digit #90, a 3.

MNIST (LeCun, Cortes and Burges), grey levels scaled to [0, 1].

▶ Notebook 3.1

Our 3 is 784 numbers between 0 and 1



0	0	0	0.3	0.8	1.0	1.0	1.0	1.0	1.0	1.0
0	0.0	0.5	1.0	1.0	0.9	0.6	0.3	0.2	0.5	1.0
0	0.4	1.0	0.9	0.6	0.1	0	0	0.0	0.5	1.0
0	0.1	0.6	0.1	0	0	0	0	0.2	1.0	1.0
0	0	0	0	0	0.0	0.4	0.6	0.9	1.0	0.9
0	0	0	0	0	0.3	1.0	1.0	1.0	0.9	0.2
0	0	0	0.6	0.9	1.0	1.0	1.0	1.0	1.0	0.4
0	0	0	0.9	1.0	1.0	1.0	0.7	1.0	1.0	1.0
0	0	0	0.0	0.2	0.0	0.0	0	0.2	0.7	1.0
0	0	0	0	0	0	0	0	0	0.1	0.9

Each pixel is a number: 0 is paper, 1 is full ink.

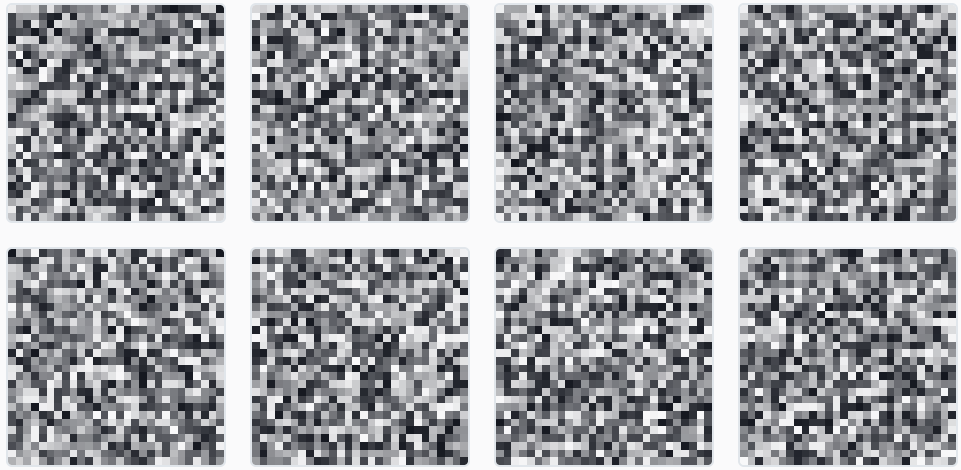
$$x \in [0, 1]^{784}$$

One image = one point in a 784-dimensional space.

> Flatten the 28 × 28 grid into a list: x is one point in a 784-dimensional space. A model of digits is a distribution over that space.

▶ Notebook 3.1

A random point in that space looks like static



Each pixel drawn uniformly between 0 and 1.

Almost every point in the space is static. Digits live in a tiny region; p(x) must find it.

> You would have to draw for longer than the age of the universe to hit a digit. A good p(x) concentrates on the tiny region where digits live.

▶ Notebook 3.2

What we fit: one distribution for all 60,000 training digits



All ten digits, mixed together. We do not use the labels.

One model p(x), not ten. Sampling it should give 0s, 1s, …, 9s, each about as often as in the data.

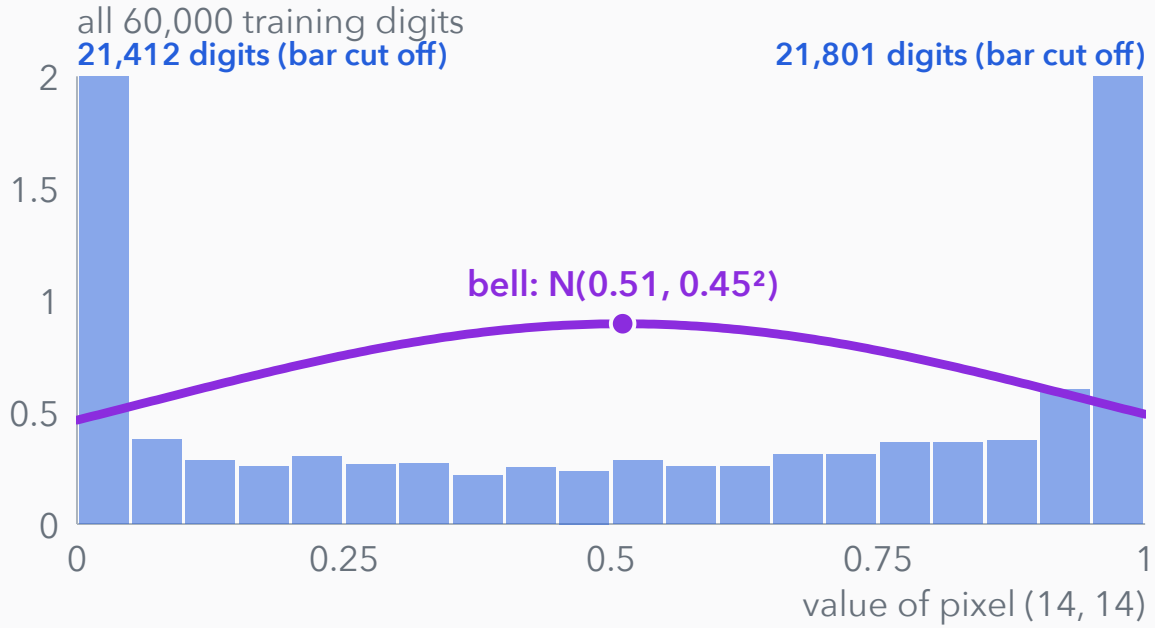
Each digit is a list of 784 numbers. p(x) must say how likely each list is.

> An image is 784 numbers, so p(x) is a distribution over lists of 784 numbers. The simplest idea: treat each of the 784 positions on its own.

MNIST training set.

▶ Notebook 3.2

Start with one pixel: position (14, 14), in every training digit



34% of digits leave it blank (0); 52% put ink on it (above 0.5).

$$m = \frac{1}{60000}\sum_i x_i = 0.51, \quad s = 0.45$$

The bell peaks at a grey that few digits have: Part 2's average-is-a-poor-sample, again. We use it anyway; it is the simplest choice.

Fit one Gaussian to these 60,000 numbers, exactly as in Part 2: mean m = 0.51, spread s = 0.45.

MNIST training set. Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 3.2

Do it for all 784 pixels, then sample each pixel from its own bell

8 samples



Repeat the one-pixel fit at every position: 784 means and 784 spreads, 1,568 numbers. That is the whole model.

To sample, every pixel draws from its own bell, without looking at any other pixel:

```
x = (m + s * torch.randn(784)).clamp(0, 1)
```

Takeaway. Each pixel decides alone, so no strokes form: the samples are speckle around the average digit.

The result: grey speckle. A stroke needs neighbouring pixels to be dark together, and this model never lets them coordinate.

MNIST training set. Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 3.2

Why it fails: neighbouring pixels move together



correlation of the centre pixel (14, 14) with every pixel: teal positive, red negative

pixel pair	correlation
centre and its right neighbour	0.78
two to the right	0.29
five to the right	−0.21
a far corner pixel	−0.01

Takeaway. Pixels are not independent. Modelling every pair directly needs a 784 × 784 covariance: over 300,000 numbers. We need a smarter idea.

> Strokes are made of neighbouring pixels that move together. A model of digits must capture that.

Correlations over the 60,000 MNIST training digits. Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 3.3

Digits vary in only a few ways



35 test 3s, placed by two numbers computed from all 6,131 training 3s.

Neighbours look alike. Left to right the strokes get thicker; top to bottom the 3s become more upright and rounder.

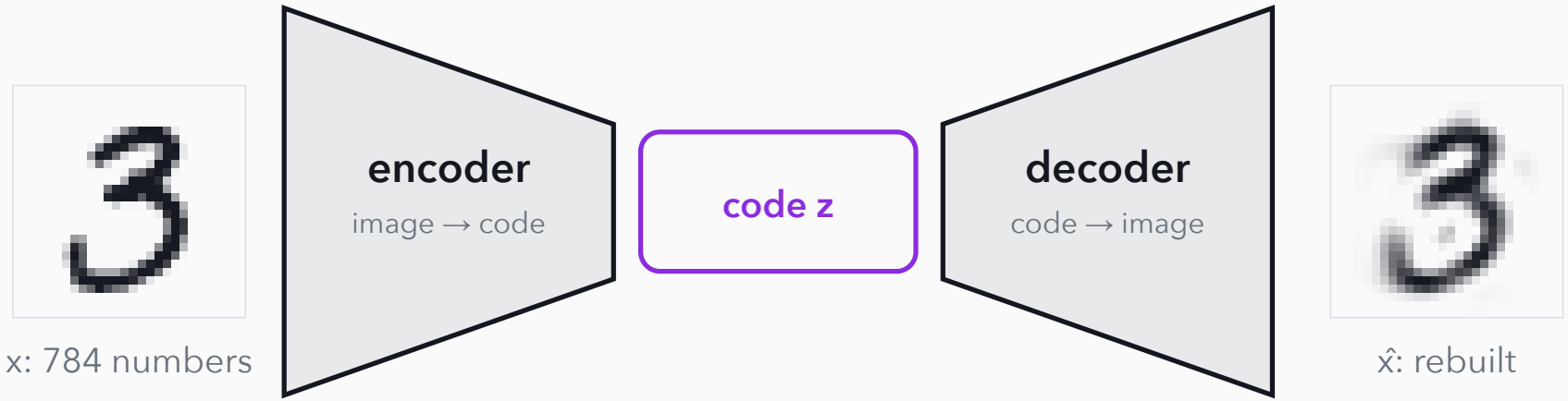
> **Takeaway.** A 3 is not 784 independent numbers. A few numbers (a **code**) can say which digit and which style.

So describe each image by a short code instead of 784 pixels, and model the codes. That is the plan for the rest of the lecture.

MNIST 3s. Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 3.3

Encoder and decoder: image → a few numbers → image

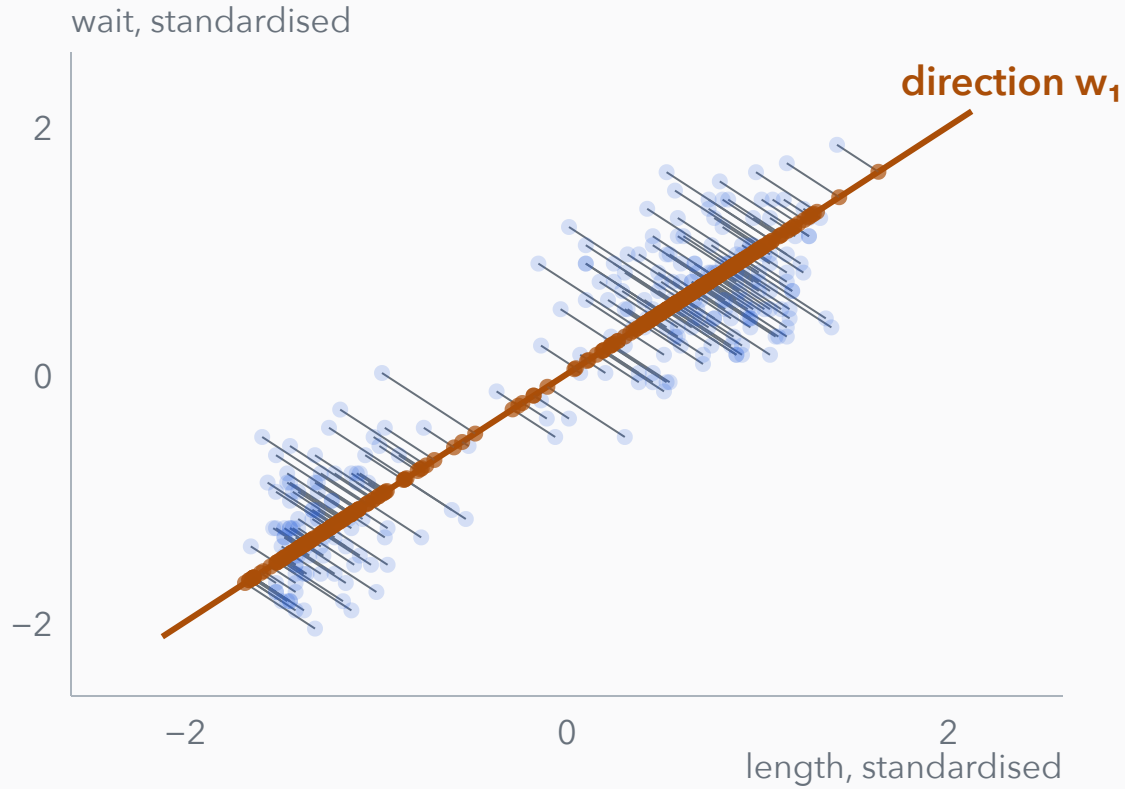


Takeaway. If the rebuilt image matches the original, the code kept what matters. The simplest encoder and decoder are straight lines: PCA.

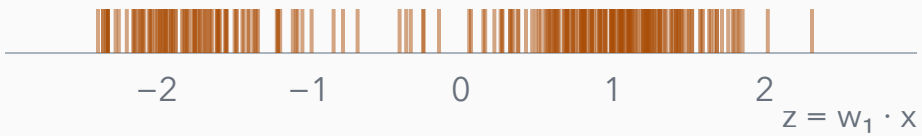
Train or choose both so that the image comes back. With straight-line maps this is PCA; with neural networks (Part 4) it is an autoencoder.

▶ Notebook 3.3

PCA, the simplest encoder and decoder, in two dimensions



encode: one number per eruption



share of the spread along w_1: 95%

the other direction keeps 5%

decode: $\hat{x} = z \cdot w_1$ (orange)

two numbers per eruption became one

Decode: go back from the number to its point on the line, close to where it started. That is PCA.

Old Faithful geyser data (Azzalini and Bowman, 1990), as distributed with R (`datasets::faithful`): 272 eruptions.

▶ Notebook 3.3

PCA on 60,000 digits: a mean image and directions



$$\underbrace{z = W^\top (x - m)}_{\text{encode: } 784 \to d} \qquad \underbrace{\hat{x} = m + Wz = m + z_1 w_1 + \cdots + z_d w_d}_{\text{decode: } d \to 784}$$

> Encode: d numbers per image, z = $W^T(x - m)$. Decode: start from the mean and add z_1 times w_1, z_2 times w_2, and so on.

PCA of the 60,000 MNIST training digits (eigenvectors of the 784 × 784 covariance). Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 3.3

Rebuild our 3 from its first d PCA numbers



our 3: 784 numbers

→

10 numbers

$z_1 = 0.88, z_2 = 2.55, \ldots$

→



rebuilt from the first 10

26.4

squared error

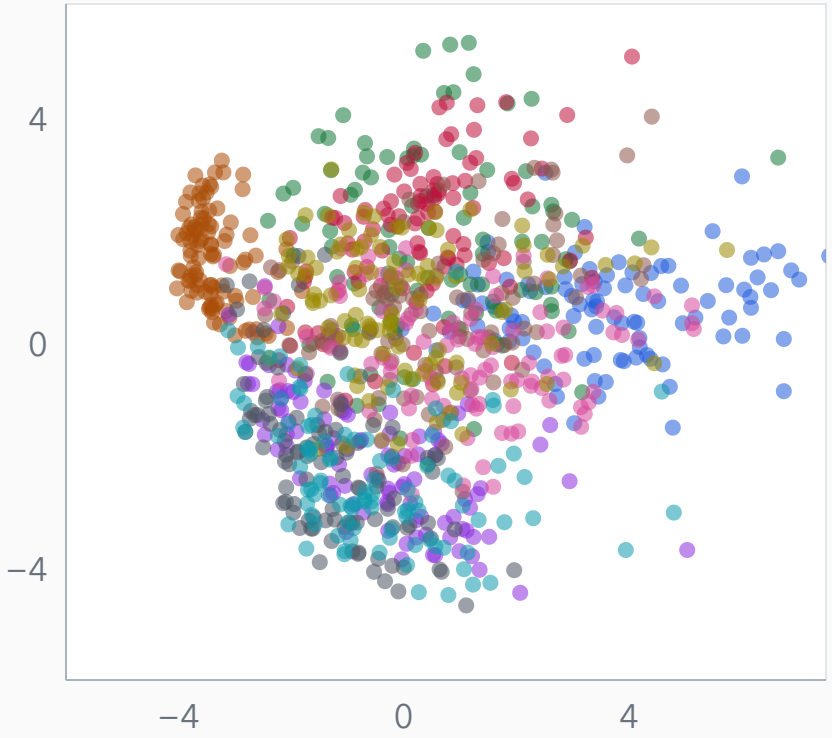
48.8%

of the variance of all 60,000 digits kept

Drag d: 2 numbers give a blur (error 41.3), 10 a rough 3 (26.4), 50 a clear 3 (7.2).

▶ Notebook 3.3

Each digit as 2 numbers: PCA's map of 1,000 test digits



0 1 2 3 4 5 6 7 8 9

z_1, z_2 from PCA

hover any point to see its digit

> 0s and 1s separate; 4, 7 and 9 overlap heavily. Two straight-line numbers cannot pull every digit apart.

1,000 MNIST test digits, 100 per class (seed 2024). Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

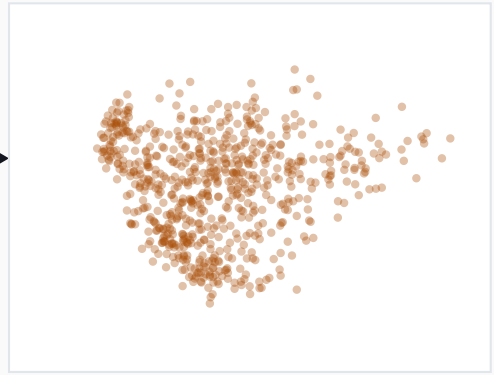
▶ Notebook 3.4

To generate, four steps: encode, fit, draw, decode

① encode the training digits

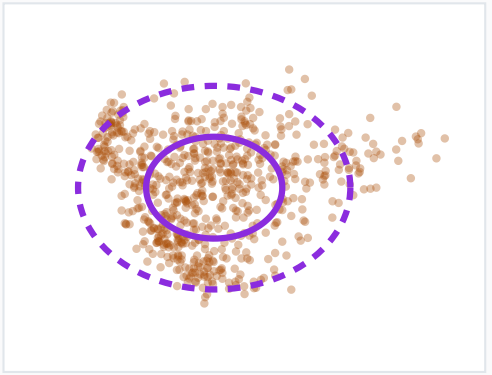


60,000 digits …



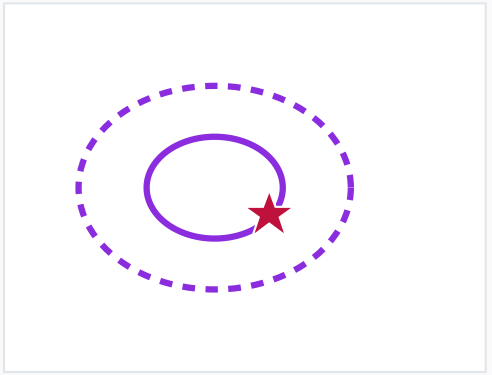
… become 60,000 dots (codes)

② fit a bell to the dots



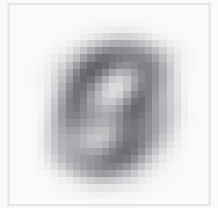
p(z): Part 2, in 2-D

③ draw a new dot



z = (1.81, −0.97)

④ decode it



m + W z

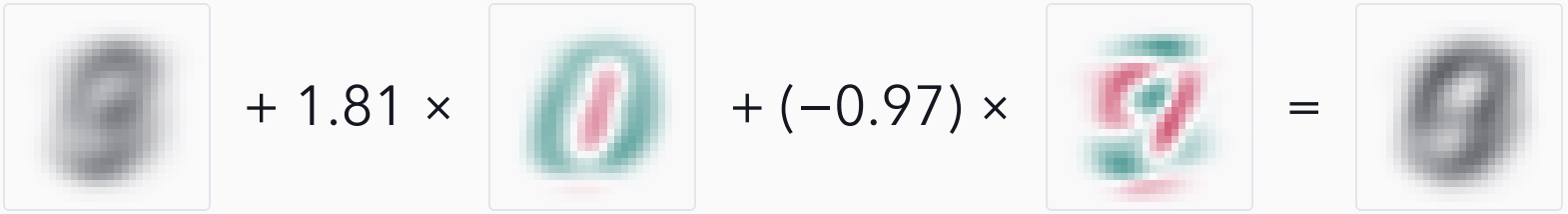
Takeaway. Steps ① and ② fit the model, once. Steps ③ and ④ make a digit, every time. The code is hidden first, then the image: the same shape as the geyser's kind, then wait.

Fitting happens once; sampling repeats steps 3 and 4. Next: step 3 and 4 by hand, with numbers.

▶ Notebook 3.4

Steps ③ and ④ by hand: first a code, then an image

$$\epsilon = (0.8,\ -0.5) \sim \mathcal{N}(0, I) \quad \Rightarrow \quad z = \big(2.26 \times 0.8,\ 1.93 \times (-0.5)\big) = (1.81,\ -0.97)$$



mean image m, plus z_1 times direction w_1, plus z_2 times direction w_2 (teal adds ink, red removes it)

A new image, drawn in two steps: the parent z first, then the child x. Repeat with fresh ε for more samples.

> The result is a sample: a blurry digit, because two PCA numbers can only describe a blur. More numbers and more bumps come next.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 3.4

A mixture of K Gaussians over d numbers: try it



d = 20 PCA numbers · K = 10 bumps · 2,309 fitted numbers · held-out average log-likelihood −23.9

> Choose d and K, then press Sample: 16 codes from the mixture, each decoded with PCA. Compare K = 1 and K = 20 at d = 20.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 3.4

What limits the PCA recipe?



PCA's decoder is a straight-line map: $\hat{x} = m + W z$.

Bend it: replace both maps with small neural networks. That is an autoencoder.

> A flexible encoder and decoder should keep more from the same few numbers. Part 4.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 3.4

Part 3 in one slide

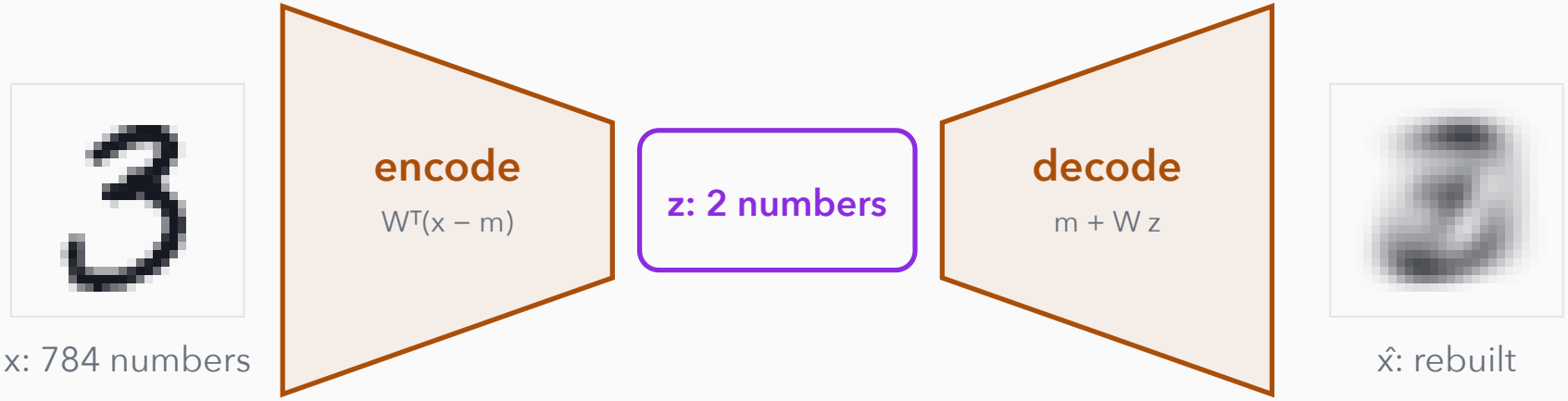
model	numbers	samples
independent pixel Gaussians	1,568	speckle
one full Gaussian	308,504	ghosts
Gaussian on 50 PCA numbers	1,325	blended digits
20 bumps on 20 PCA numbers	4,619	rough but real digits

Compress first, then fit: the code is the hidden variable. Next, learn the compression itself.

> The weak link is PCA's straight-line decoder. Part 4 learns a better one.

▶ Notebook 4.1

PCA is an encoder and a decoder

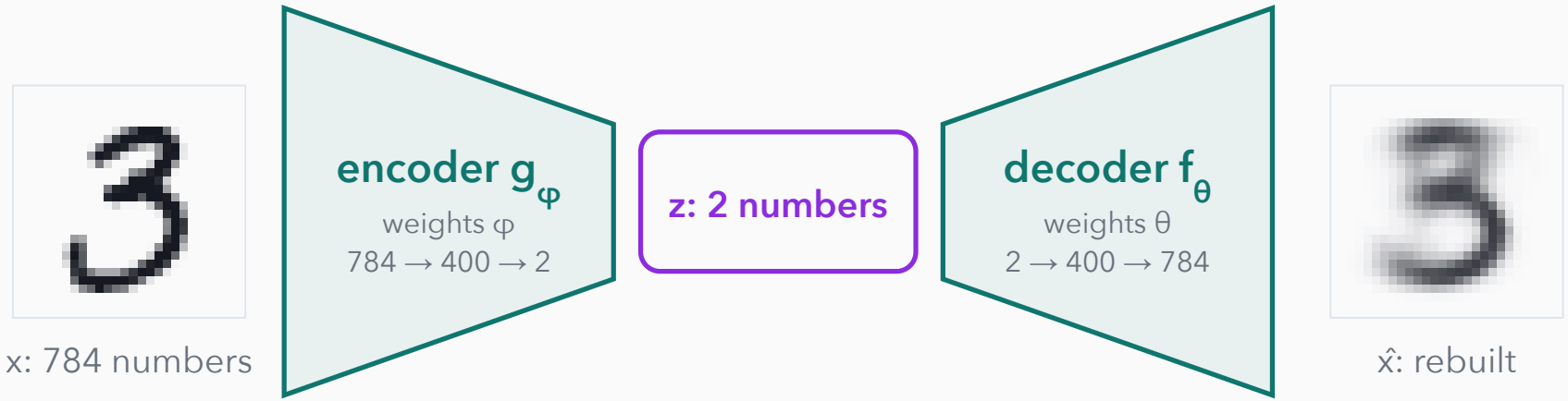


Both maps are straight lines (matrix products), chosen to make the rebuilt image close to the original.

> PCA's two maps are linear. For our 3 with 2 numbers the squared error is 41.3.

▶ Notebook 4.1

An autoencoder: replace both maps with small neural networks



The decoder f_θ, with weights θ, mirrors it: 2 → 400 → 784, then a sigmoid. Train φ and θ together so that $\hat{x} \approx x$.

▶ Notebook 4.1

Two small networks in PyTorch


```python
class AutoEncoder(nn.Module):
    def __init__(self, z=2):
        super().__init__()
        self.encoder = nn.Sequential(  # g_φ, weights φ
            nn.Linear(784, 400), nn.ReLU(),
            nn.Linear(400, z))
        self.decoder = nn.Sequential(  # f_θ, weights θ
            nn.Linear(z, 400), nn.ReLU(),
            nn.Linear(400, 784), nn.Sigmoid())

    def forward(self, x):
        z = self.encoder(x)     # 784 -> z numbers
        return self.decoder(z)  # z -> 784 numbers
```

Why a sigmoid at the end? Are we classifying?

No: it keeps every output pixel between 0 and 1, like the data.

> The final sigmoid squashes each of the 784 outputs into (0, 1), the range of our pixels.

▶ Notebook 4.1

The loss: squared error, summed over pixels, averaged over images

$$L(\phi, \theta) = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{784} \left(x_{ij} - \hat{x}_{ij} \right)^2, \qquad \hat{x}_i = f_\theta\left(g_\phi(x_i)\right)$$

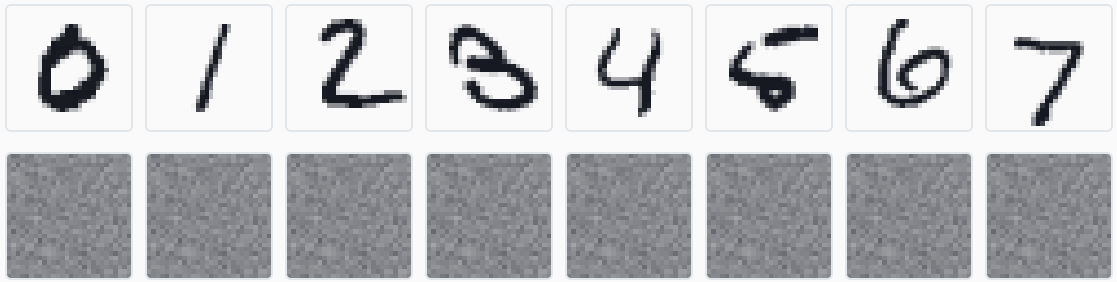
Our 3, through a trained 2-number autoencoder: squared error **26.3**.

PCA on the same 3 with the same 2 numbers: **41.3**. Same budget, bent maps, less error.

> PCA, the best straight-line version, leaves 41.3. The bend buys a lot.

▶ Notebook 4.2

Before training: always look at what the network outputs



Every output pixel is between 0.38 and 0.66: flat grey, whatever the input.

Each pixel is off by about 0.5: the target is 0 (paper) or 1 (ink), the output about 0.5. Squared: about 0.24.

$$\underbrace{784}_{\text{pixels}} \times \underbrace{0.235}_{\text{per pixel}} \approx 185$$

Takeaway. 185 is where training starts. A trained model should bring it far down; if it does not move, something is broken.

Always check this before training: it tells you the starting loss and catches bugs early.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 4.2

One training step: forward, loss, backward, update


```
model = AutoEncoder(z=2)
opt = torch.optim.Adam(model.parameters(), lr=1e-3)
for epoch in range(30):
    for x in loader:          # 128 images at a time
        x_hat = model(x)
        loss = ((x_hat - x)**2).sum(1).mean()
        opt.zero_grad()
        loss.backward()       # encoder AND decoder
        opt.step()
```

469 steps per pass over the 60,000 digits; 30 passes.

One loss, one backward pass: the gradient flows from the error back through the decoder, the code, and the encoder.

> loss.backward() sends gradients through the decoder, into the code, and on into the encoder: both are trained together.

▶ Notebook 4.2

Press play: reconstructions sharpen as it trains

8 test digits



what it outputs after 14,070 steps (30 epochs)

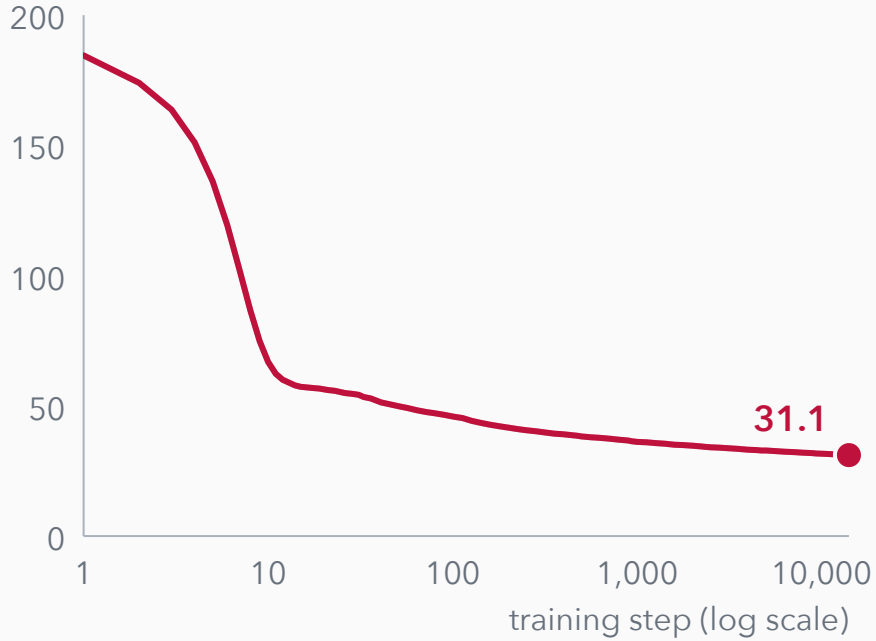


28	2	54	47	23	70	54	12

squared error of each digit (summed over its 784 pixels)

average over the 10,000 test digits: 31.1

test error while training



> The error keeps falling as each digit takes shape; after 30 epochs it is 31.1. The number under each digit is that digit's own error.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 4.2

Did it memorise? Compare training and test error

2-number code	training error	test error
autoencoder	30.3	31.1
PCA		43.6

Test digits were never used for training.

Training and test errors are close: it learned digits in general, not these particular 60,000.

> The gap is small (30.3 against 31.1). And the autoencoder beats PCA by a wide margin with the same 2 numbers.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 4.2

Take out the ReLUs and the sigmoid: a linear autoencoder rediscovers PCA



Test error: linear autoencoder 43.8, PCA 43.6.

Its decoder spans almost the same plane as PCA's two directions (cosines of the angles between them: 1.000 and 0.92).

> Gradient descent found PCA's plane on its own. The autoencoder is PCA with bendable maps.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 4.3

Wider codes reconstruct better



Seven autoencoders, codes of 2 to 128 numbers, trained the same way.

2 numbers: 31.1. 128 numbers: 1.2.

More numbers in the code, less error: with 128 numbers each digit is rebuilt almost exactly.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 4.3

The same 8 test digits through codes of 2, 8, 32 and 128 numbers

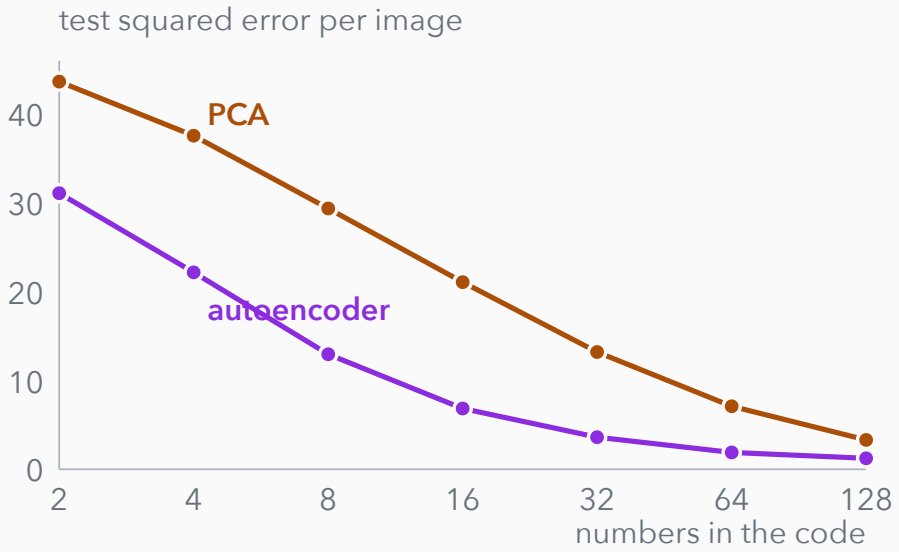


> 32 and 128 numbers reproduce strokes almost exactly.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 4.3

Autoencoder against PCA, at every code size



8 numbers: PCA 29.4, autoencoder 12.9.

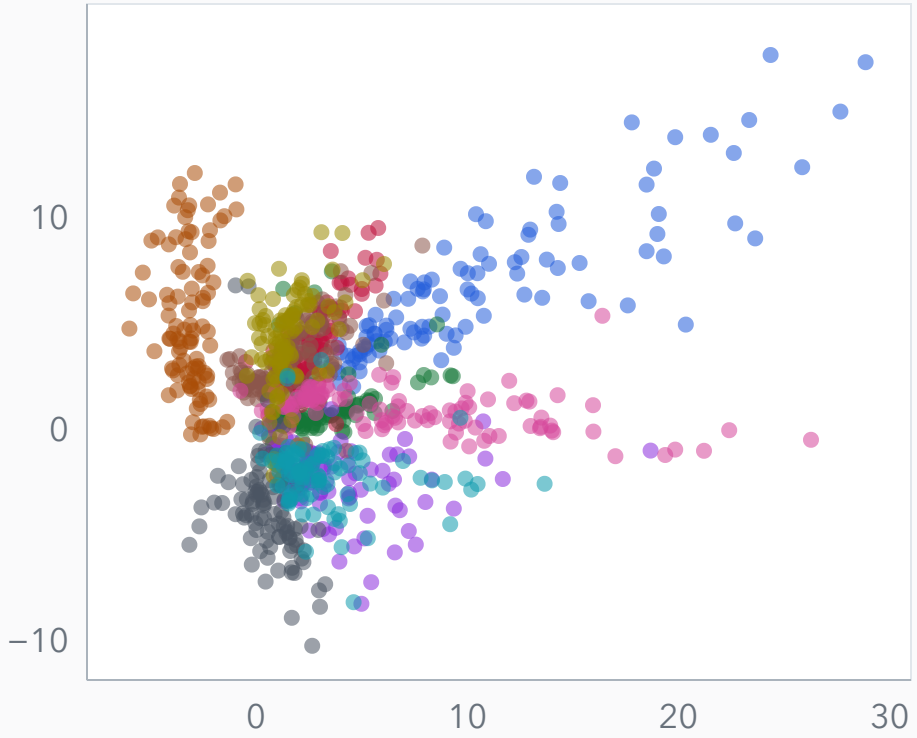
PCA needs about four times as many numbers for the same error.

> The biggest gain is for short codes, where bending matters most.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 4.4

Where the codes land: the autoencoder with a 2-number code



0 1 2 3 4 5 6 7 8 9

its whole code: z_1 and z_2

This autoencoder has exactly 2 numbers in its code, so nothing is hidden here. Hover any point to see its digit.

> Compared with PCA's map, the digits form tighter, better-separated groups. No labels were used in training.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 4.4

Part 4 in one slide

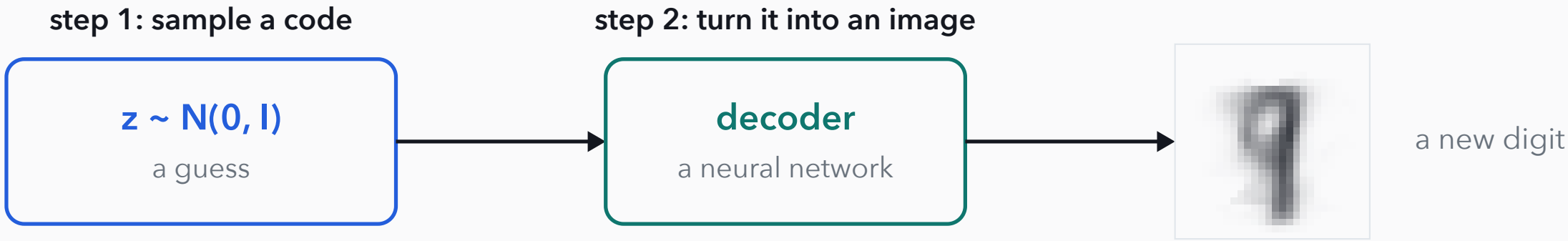
	PCA	autoencoder
encoder, decoder	straight lines	small networks
trained by	an eigenvector formula	gradient descent on squared error
2-number test error	43.6	31.1
codes	overlapping	tighter groups

PCA gave us generation by “sample a code, decode it”. Can the autoencoder’s decoder do the same?

> Part 5: keep only the decoder, feed it random codes, and see what it draws.

▶ Notebook 5.1

Keep the decoder, throw away the encoder, feed it random numbers

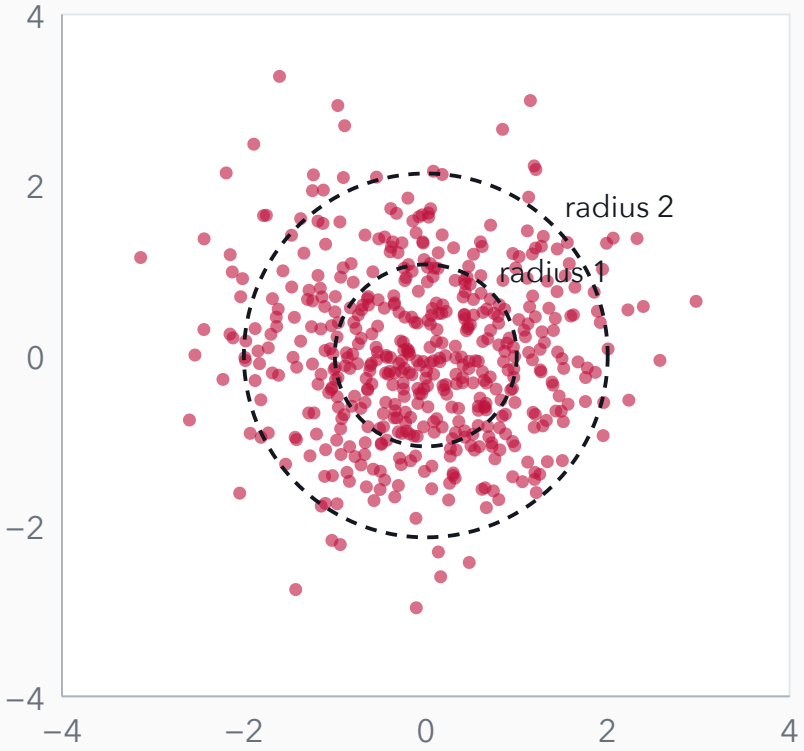


Which random numbers? The simplest choice: z ~ N(0, I).

> We need a distribution to draw codes from. Start with the simplest: the standard normal, N(0, I).

▶ Notebook 5.1

N(0, I) in two dimensions: two independent standard bells



$$z = (z_1, z_2)$$

$$z_1 \sim \mathcal{N}(0,1), \quad z_2 \sim \mathcal{N}(0,1), \text{ independent}$$

Of these 500 draws, 39% land within distance 1 of the origin and 88% within 2 (in theory 39% and 86%).

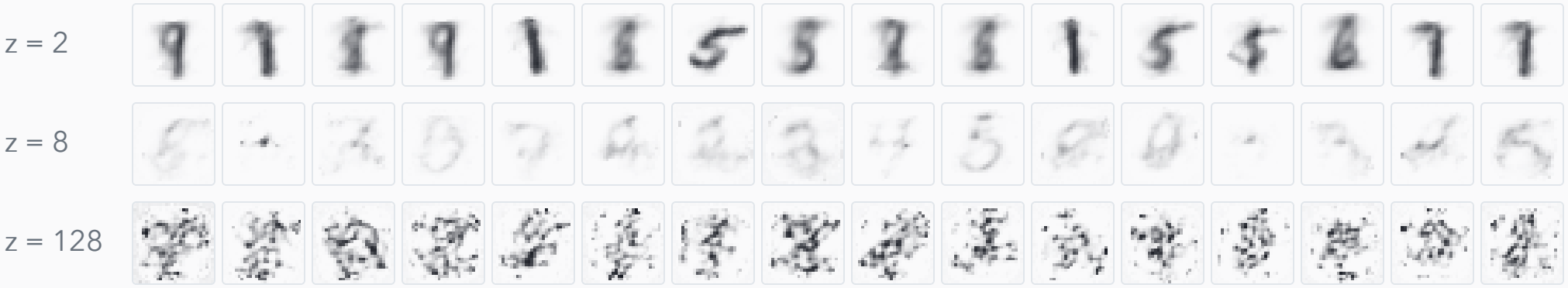
In code: `torch.randn(500, 2)`. Draws pile up near (0, 0) and rarely go beyond 3.

That is all torch.randn does. Its draws live in a small disc around (0, 0).

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 5.1

Random codes through three autoencoders



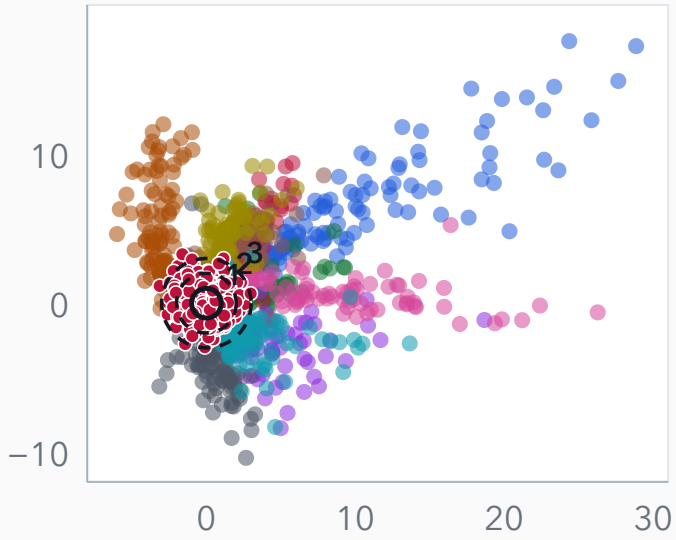
Takeaway. Better reconstruction did not give better samples: the best reconstructor makes static. Why? Look at where the codes are.

128 numbers, the best reconstructor of all: static. Better reconstruction, worse generation.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 5.2

Put the random codes on top of the real ones



0 1 2 3 4 5 6 7 8 9

real codes and 500 random ones (red)

Circles: N(0, I) at radius 1, 2 and 3.
Hover a dot to see the digit and what the decoder draws from that code.

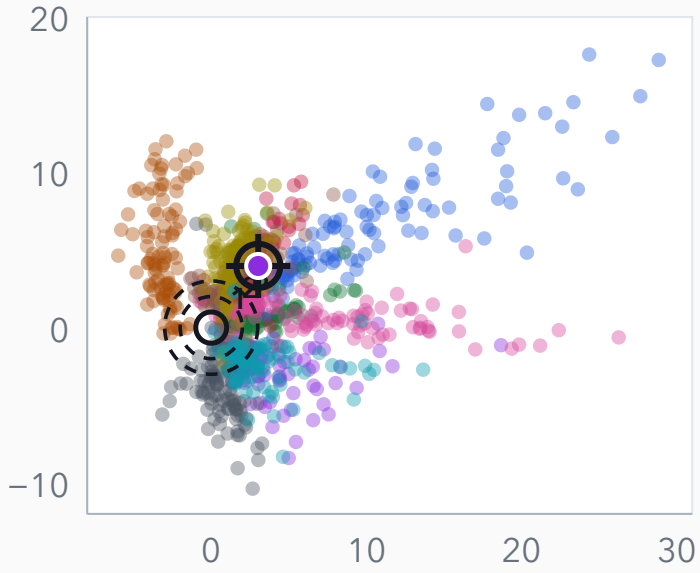
Takeaway. N(0, I) covers only a small corner of the real codes: random codes reach a few digit types and miss the rest.

Nothing in the loss asked the codes to sit near N(0, I). The red draws land in one corner: some digits there, nothing of the rest.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 5.2

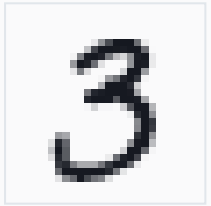
Your turn: three ways to choose a code



0 1 2 3 4 5 6 7 8 9

decoder of the autoencoder, run in your browser





the digit we encoded

circles: N(0, I)
at radius 1, 2, 3

z = (3.02, 3.97)
the code of our 3, decoded

Takeaway. Encoded real digits decode well. Random N(0, I) codes and clicks in empty space mostly do not.

Only codes near real codes give good digits, and N(0, I) does not know where those are.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 5.2

Decode a whole grid of codes





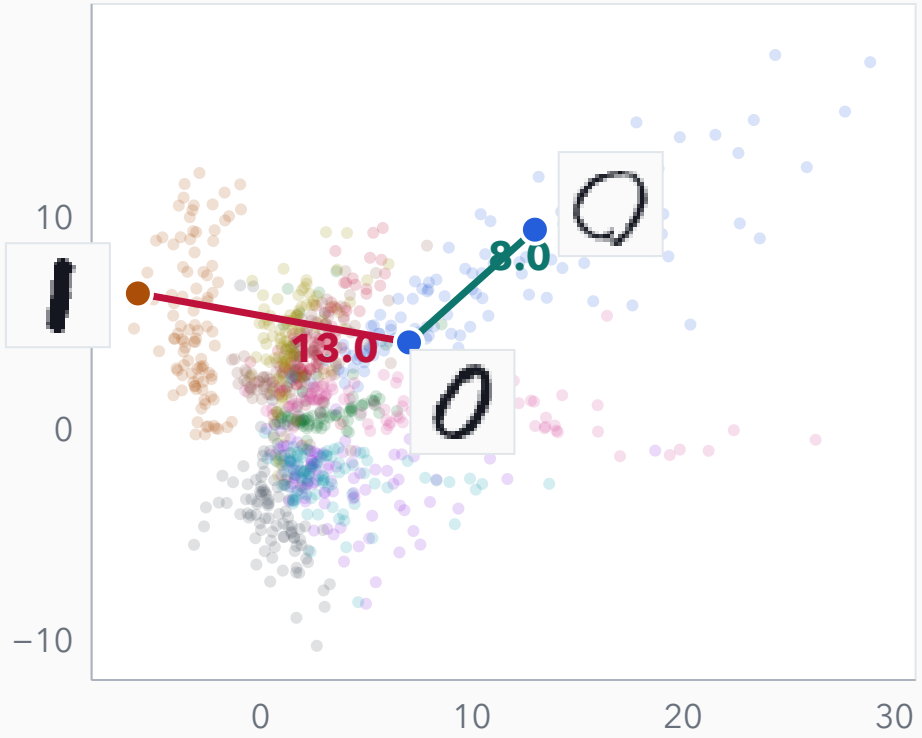
Takeaway. Good digits only where real codes are. Away from them the decoder only extrapolates.

Over the real codes the grid shows digits blending into each other. Far from them the decoder only extrapolates: thick 1s and 0s, and broken strokes in the bottom-left corner.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 5.2

How close are codes of similar digits? Measure it



Two 0s and a 1 from the test set, at their 2-number codes.

Distance between the two 0s (teal).

Distance from a 0 to the 1 (red).

Averaged over all pairs among 100 test 0s and 100 test 1s: two 0s are 8.0 apart, a 0 and a 1 are 14.7 apart.

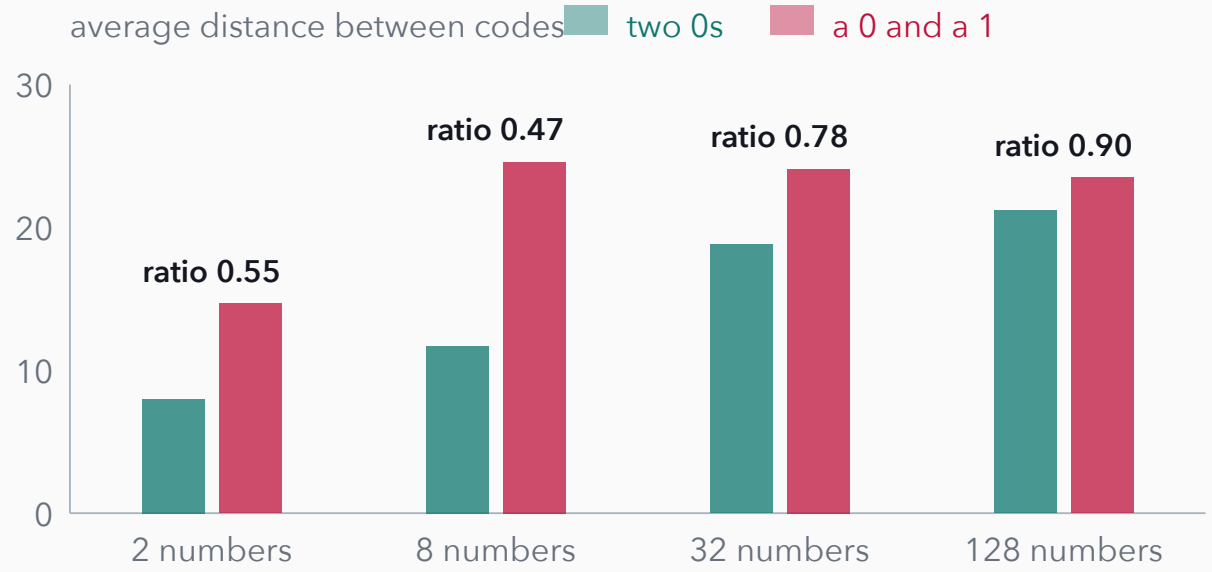
> **Takeaway.** Ratio 0.55: with 2 numbers, 0s sit closer to each other than to 1s. Similar digits get similar codes.

> Average over many pairs and take the ratio: below 1 means same-digit codes are closer than different-digit codes.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 5.2

Wider codes: similar digits drift apart



The same measurement for autoencoders with 2, 8, 32 and 128 numbers in the code.

Takeaway. With 128 numbers two 0s are almost as far apart as a 0 and a 1 (ratio 0.90). The loss only rewards reconstruction, so every digit gets its own private spot, and the space between them is empty.

Nothing in the loss keeps codes near each other or near N(0, I). Great for reconstruction, bad for generation: this is what the VAE will fix.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 5.2

Walk in a straight line from a 1 to a 0



Takeaway. Straight lines between codes leave the region where real codes live. The space between digits is not filled with digits.

With 16 numbers the endpoints are sharp, but the middle passes through blobs that are no digit: the straight line leaves the region where codes live.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 5.3

What we saw, and what we want instead

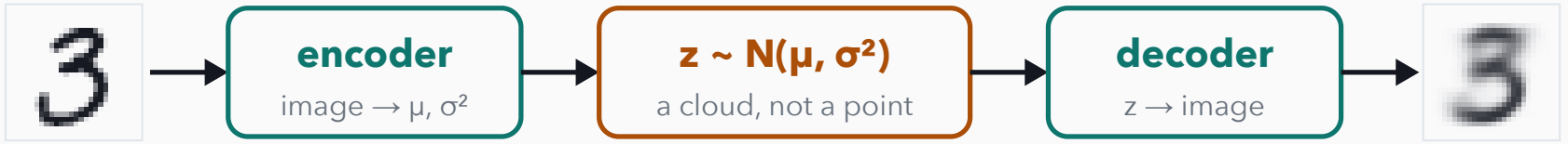
what we saw	what we want	how the VAE does it
random N(0, I) codes miss the real codes	codes that fill N(0, I), so `torch.randn` gives good codes	a KL penalty pulls every code towards N(0, I)
similar digits drift apart; gaps decode to junk	each image owns a small cloud of codes, and clouds overlap	the encoder outputs a mean and a spread; z is drawn from that cloud
no way to say how likely an image is	one probability model that can score and sample	a prior and a decoder, trained with Lecture 1's ELBO

> Problem three: the autoencoder cannot score an image. Want: a probability model. The variational autoencoder does all three.

▶ Notebook 6.1

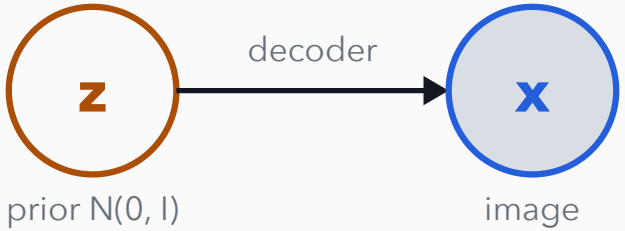
Two roads to the VAE

Road 1 · start from the autoencoder



+ a KL penalty that keeps the clouds near N(0, I)

Road 2 · start from a generative model



learn the decoder ⇒ need p(z | x), the posterior, which has no formula ⇒ approximate it with q(z | x) (variational inference, Lecture 1): the encoder.

Both roads arrive at the same network and the same loss:

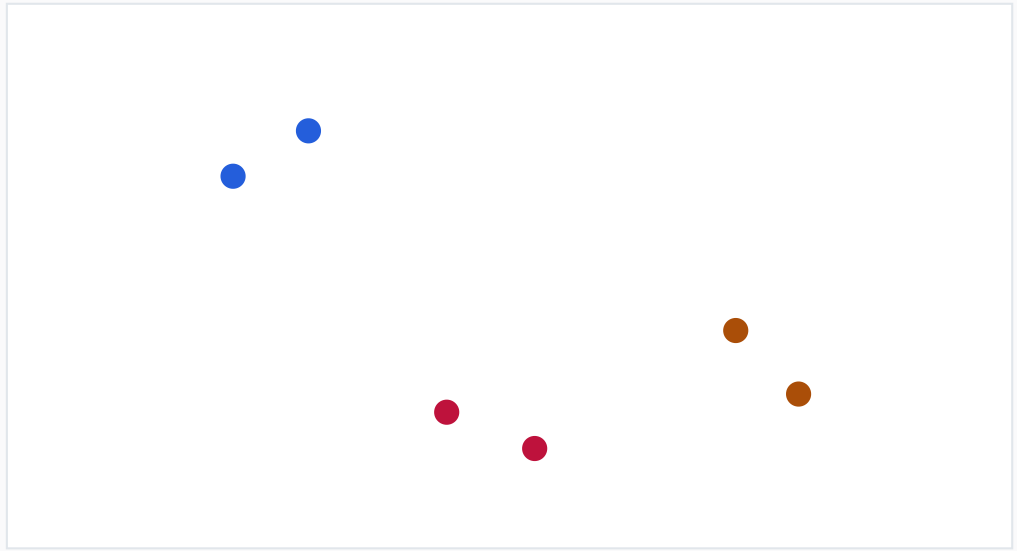
reconstruction + KL

> They meet: the same encoder, the same decoder, the same loss. We take Road 1 first, then Road 2 explains why it is the right thing to do.

▶ Notebook 6.2

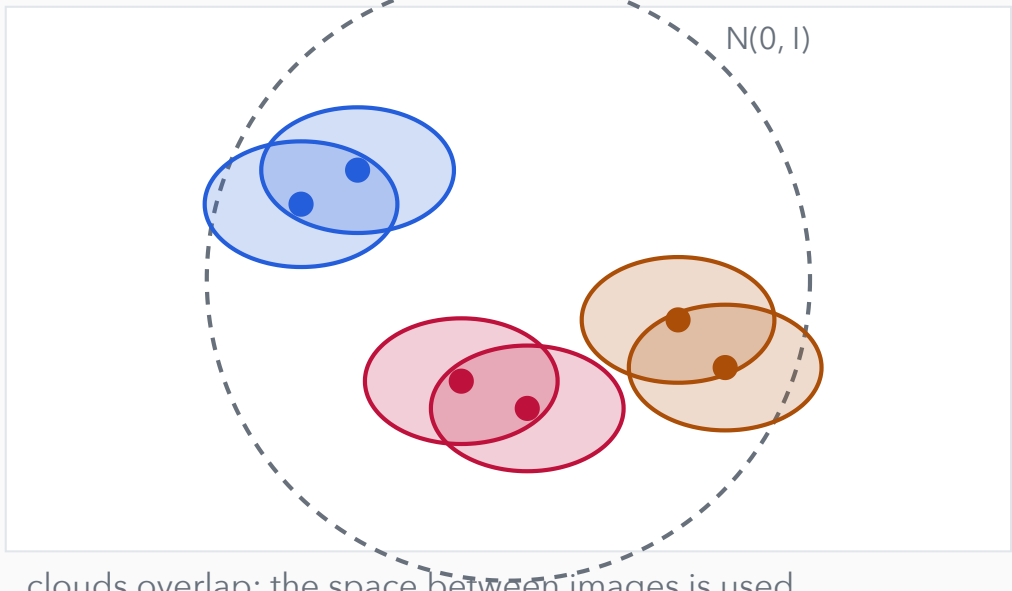
Road 1: encode each image to a cloud, not a point

autoencoder: each image → one point



points: the space between them is never trained

VAE: each image → a small cloud



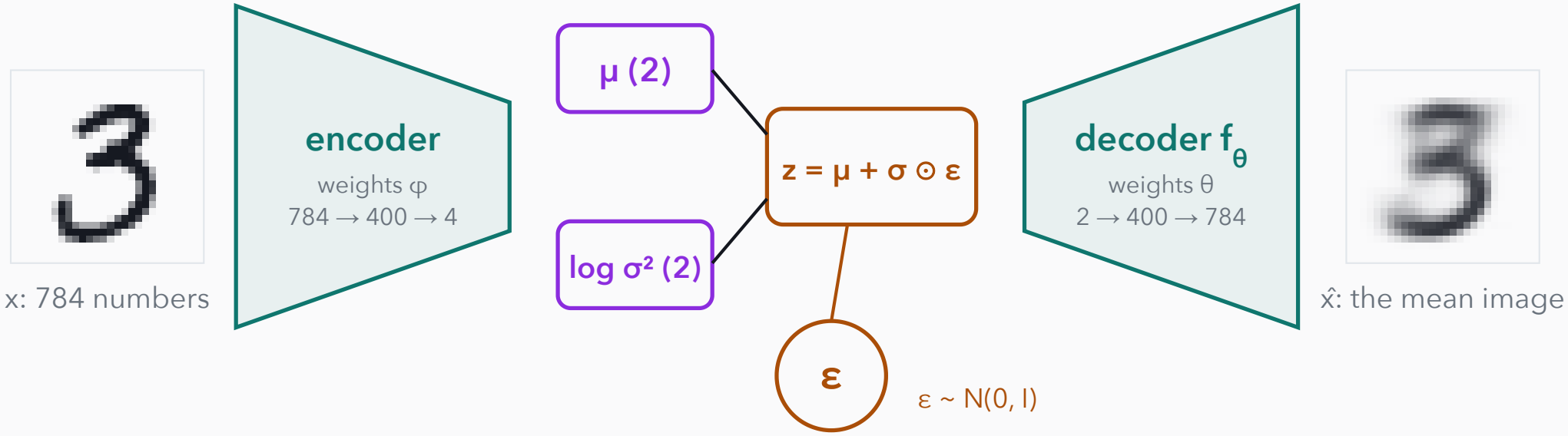
clouds overlap: the space between images is used

A sketch, not data. Colours: three kinds of digit.

> Instead, send each image to a small cloud, a Gaussian $N(\mu, \sigma^2)$, and decode a random point from the cloud. Neighbouring clouds overlap, so the gaps get trained too.

▶ Notebook 6.2

Road 1: the network, with a cloud in the middle



The decoder is unchanged: from z to a mean image.

▶ Notebook 6.2

Road 1: keep the clouds near N(0, I) with a KL penalty

$$L = \underbrace{\|x - f_\theta(z)\|^2}_{\text{rebuild the image}} + \underbrace{\mathrm{KL}\big(\mathcal{N}(\mu, \sigma^2) \,\|\, \mathcal{N}(0, I)\big)}_{\text{keep the cloud near } \mathcal{N}(0,I)}$$

$$\mathrm{KL} = \tfrac{1}{2} \sum_k \left(\mu_k^2 + \sigma_k^2 - \log \sigma_k^2 - 1\right)$$

↗ Lecture 1: KL between diagonal Gaussians

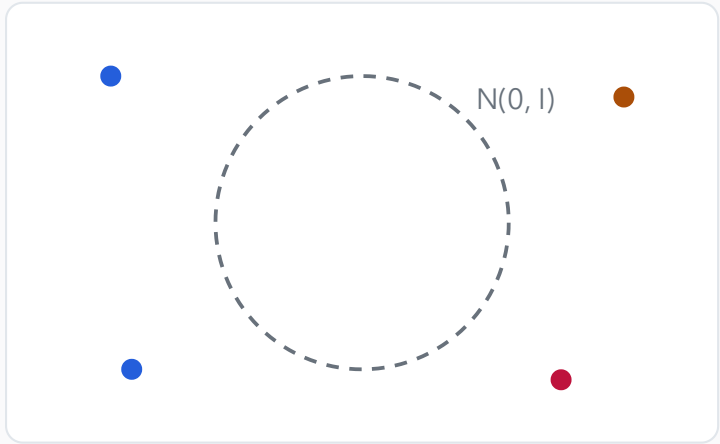
Takeaway. Without the penalty, the encoder could shrink the clouds to points and spread them far apart: an autoencoder again. The KL stops both.

The penalty pulls every cloud towards N(0, I) (wish 1) and stops it from shrinking to a point (wish 2).

▶ Notebook 6.2

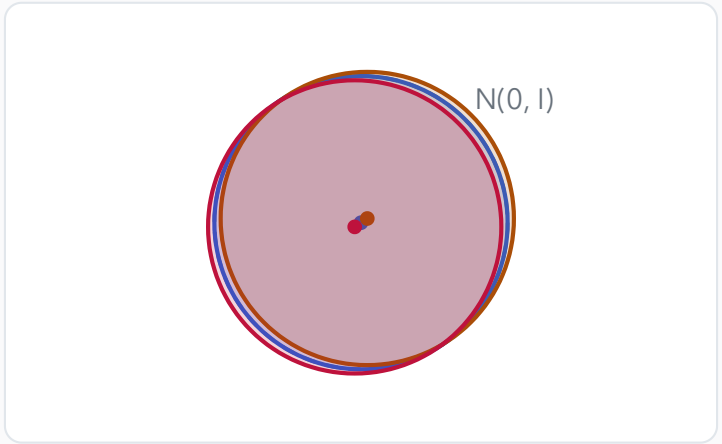
Road 1: two terms pull in opposite directions

reconstruction alone



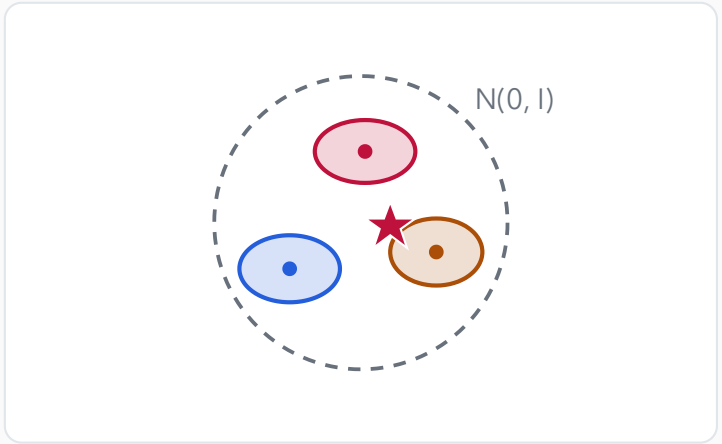
tiny clouds, far apart:
an autoencoder again

KL alone



every image gets the same cloud N(0, I):
z says nothing about the image

both together



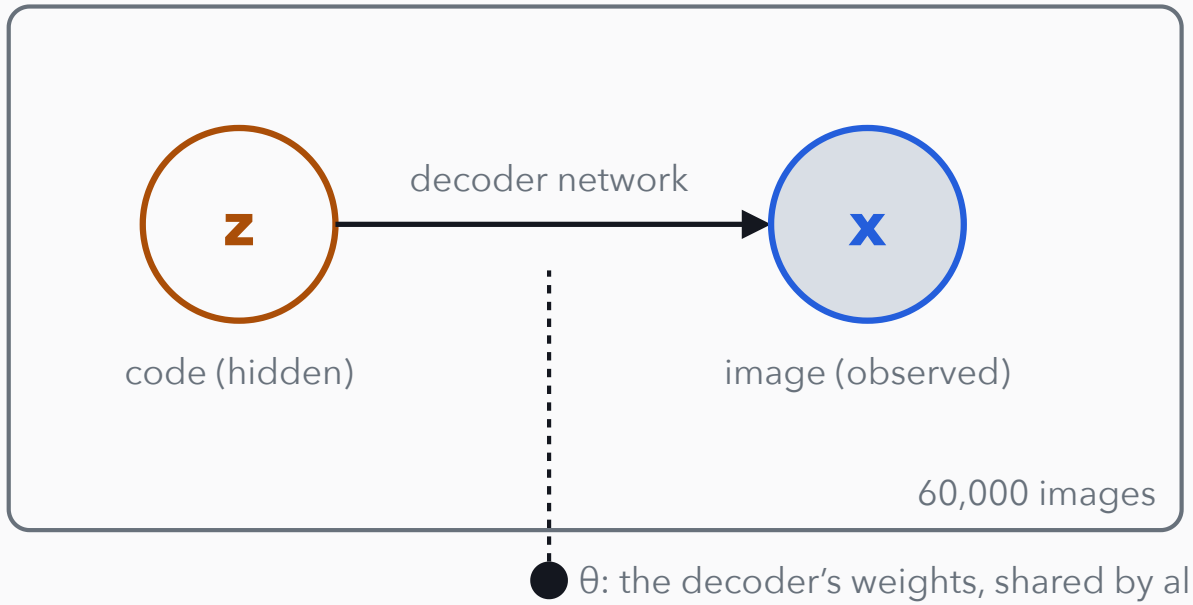
clouds near the centre, still apart:
a random z lands near some cloud

Takeaway. Road 1 in one line: an autoencoder whose codes are noisy clouds held near N(0, I). To generate, draw z ~ N(0, I) and decode.

So a random draw from N(0, I) usually lands on or near some cloud, and the decoder has been trained to draw something sensible there.

▶ Notebook 6.3

Road 2: the VAE as a generative model



this is our prior:

$p(z) = N(0, I)$, 2 numbers, fixed in advance

the likelihood:

$p(x \mid z) = N(x; \text{decoder}(z), \sigma^2 I)$

each pixel: the decoder's value + noise of size σ

> The decoder weights θ sit outside the box: one set of weights, shared by every image, and learned from the data.

▶ Notebook 6.3

To draw an image, follow the arrows: first z, then x

① draw z

```
z = torch.randn(2)
```

z = (0.82, −0.74)

→

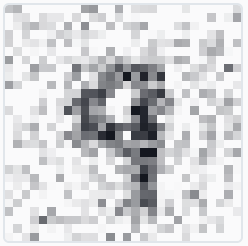
② decode



mean image $f_\theta(z)$

→

③ add pixel noise



x = mean + 0.2 × noise

Takeaway. We usually show the mean image: the pixel noise is independent speckle and adds nothing about the digit.

Step three: add independent noise of size σ = 0.2 to each pixel. That is a true sample of x; the mean image is what we display.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 6.3

The likelihood, in numbers: a squared error

$$\log p_\theta(x \mid z) = \sum_{j=1}^{784} \log \mathcal{N}\left(x_j;\ f_\theta(z)_j, \sigma^2\right)$$

$$= -\frac{1}{2\sigma^2} \left\| x - f_\theta(z) \right\|^2 \ - \ 784 \log\left(\sigma\sqrt{2\pi}\right)$$

We use σ = 0.2: about the typical pixel error of a 2-number code, √(31 / 784) ≈ 0.2. The constant term, −784 log(σ√2π), is +541.4.

> σ = 0.2 matches the size of a typical pixel error; it is a modelling choice, and Section 6.7 shows what changing its weight does.

▶ Notebook 6.3

To learn θ, maximise log p(x) for every image: an integral

$$p_\theta(x) = \int p_\theta(x \mid z)\, p(z)\, dz$$

Part 2: a sum over two kinds

$$p(x) = \sum_{k=1}^{2} \pi_k \mathcal{N}(x; \mu_k, \sigma_k^2)$$

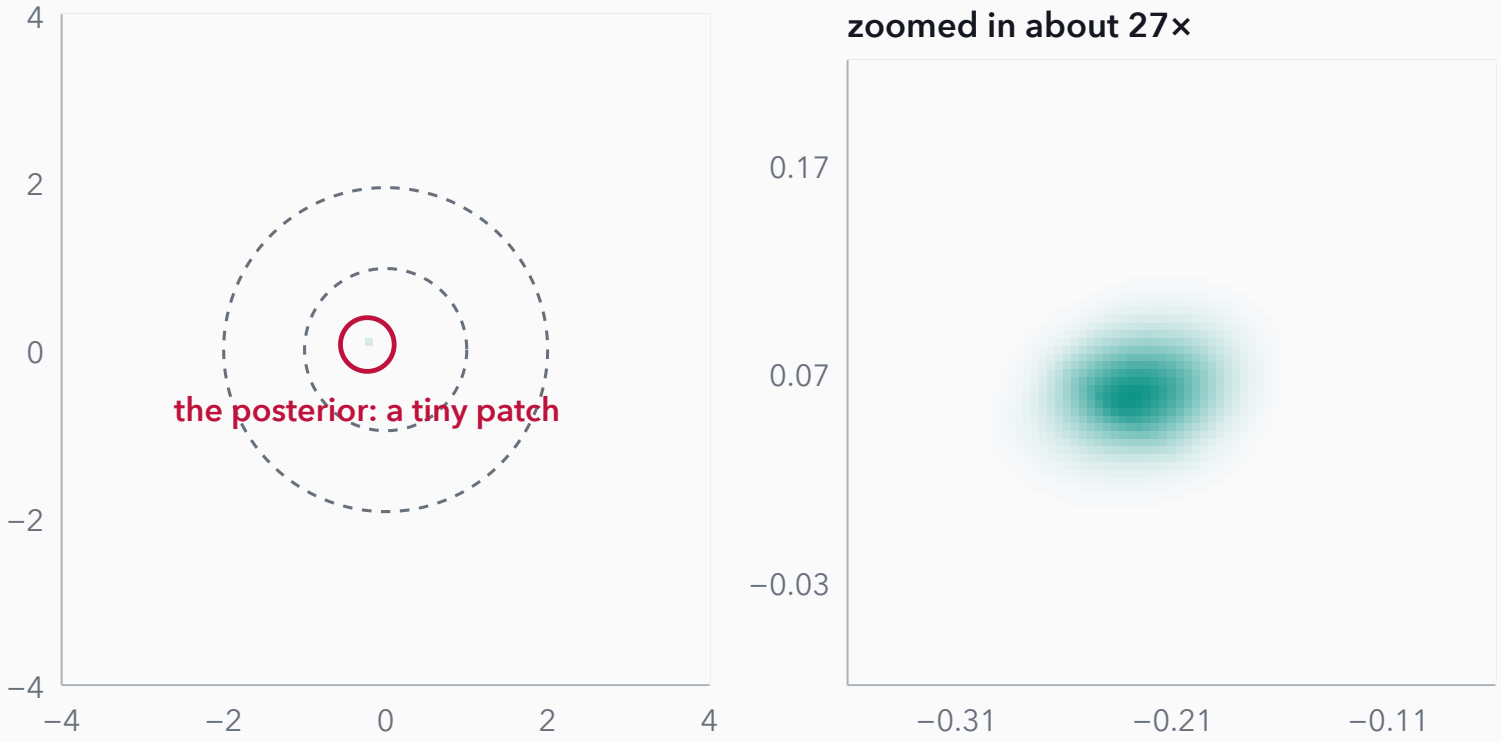
Now: an integral over every possible code

$$p_\theta(x) = \int \mathcal{N}(x; f_\theta(z), \sigma^2 I)\, \mathcal{N}(z; 0, I)\, dz$$

> For the geyser the sum had two terms. Here it is an integral over all codes, through a neural network. No formula.

▶ Notebook 6.3

For our 3: which codes could have made it?



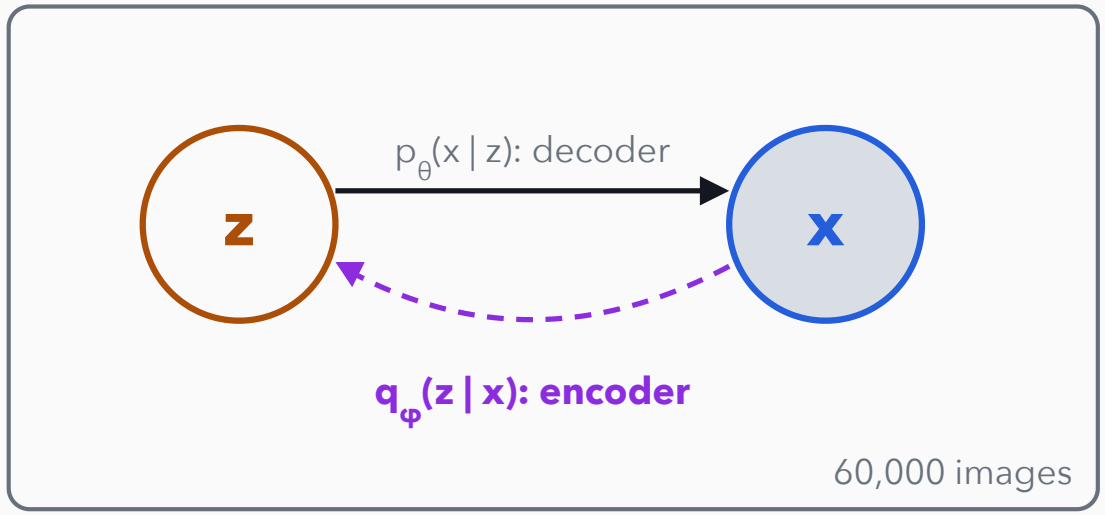
$$p(z \mid x) = \frac{p(x \mid z)\, p(z)}{p(x)}$$

> This is the posterior over codes, by Bayes' rule. Its denominator is the p(x) we want.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 6.3

The variational part: an encoder that approximates the posterior



the posterior p(z | x): which codes made this x?

Bayes' rule needs $p(x) = \int p(x \mid z)\, p(z)\, dz$: no formula

the variational part: approximate it with

$q_\varphi(z \mid x) = N(\mu_\varphi(x), \sigma_\varphi(x)^2)$, a Gaussian whose mean and spread the encoder computes from x

fit θ and φ together by maximising the ELBO

(Lecture 1, deriving the ELBO)

↗ Lecture 1: the VI idea | ↗ Lecture 1: deriving the ELBO

> Train the decoder (θ) and the encoder (φ) together by maximising the ELBO, exactly as Lecture 1 fitted q.

▶ Notebook 6.3

Road 2: where the KL comes from, for one image

$$\mathrm{KL}\big(q_\phi(z \mid x) \,\|\, p(z \mid x)\big) = \mathbb{E}_q\big[\log q_\phi(z \mid x) - \log p(z \mid x)\big]$$

$$= \mathbb{E}_q\big[\log q_\phi(z \mid x) - \log p_\theta(x \mid z) - \log p(z)\big] + \log p_\theta(x)$$

$$\log p_\theta(x) = \underbrace{\mathbb{E}_q\big[\log p_\theta(x \mid z)\big] - \mathrm{KL}\big(q_\phi(z \mid x) \,\|\, p(z)\big)}_{\text{ELBO}} + \underbrace{\mathrm{KL}\big(q_\phi(z \mid x) \,\|\, p(z \mid x)\big)}_{\geq 0}$$

Rearrange: log p(x) = ELBO + a gap that is never negative. Inside the ELBO, a KL to the prior has appeared on its own: nobody added it by hand. ↗ Lecture 1: the same derivation

▶ Notebook 6.3

The ELBO for one image is Road 1's loss

$$\text{ELBO} = \underbrace{\mathbb{E}_{q_\phi(z|x)}\big[\log p_\theta(x \mid z)\big]}_{\text{reconstruct } x} - \underbrace{\text{KL}\big(q_\phi(z \mid x) \,\|\, p(z)\big)}_{\text{stay near the prior}}$$

$$\log p_\theta(x \mid z) = -\frac{1}{2\sigma^2}\|x - f_\theta(z)\|^2 + \text{const}$$

$$-\text{ELBO} = \mathbb{E}_q\left[\tfrac{1}{2\sigma^2}\|x - f_\theta(z)\|^2\right] + \text{KL}\big(\mathcal{N}(\mu, \sigma^2) \,\|\, \mathcal{N}(0, I)\big) + \text{const}$$

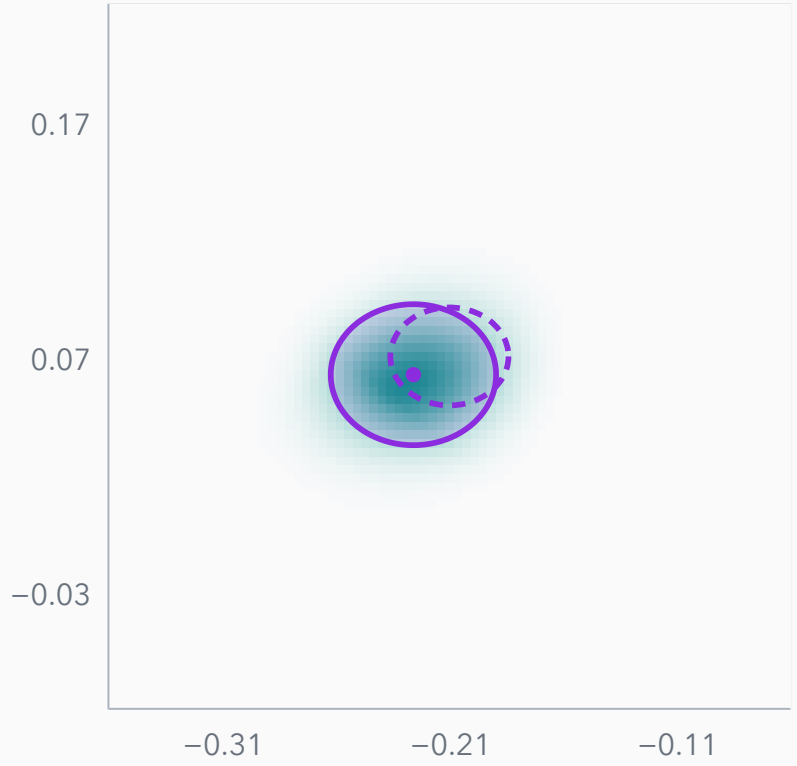
Takeaway. Road 2 gives Road 1's loss: squared error plus the KL to N(0, I). The weight on the squared error is 1 / 2σ², set by the pixel noise.

↗ Lecture 1: the ELBO in two terms

Flip the sign: squared error plus KL. The two roads meet; the penalty of Road 1 was maximum likelihood all along.

▶ Notebook 6.3

Check the encoder: its q for our 3, against the true posterior



	ELBO
log p(x), the exact target	244.07
q fitted to this image alone	244.05
q from the encoder (one pass)	243.37

Encoder: μ = (−0.211, 0.070), σ = (0.013, 0.010).

Dashed: the encoder's q. Shaded: the per-image fit. Both 2 standard deviations.

> The encoder's q sits slightly off-centre and a little too narrow. That small loss is the price of amortisation.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 6.3

Two roads, one VAE

	Road 1 · from the autoencoder	Road 2 · from a generative model
start with	an autoencoder; replace its code with a cloud	a prior $p(z) = N(0, I)$ and a decoder $p_\theta(x \mid z)$
the encoder is	a network that outputs a cloud $N(\mu, \sigma^2)$	the approximate posterior $q_\phi(z \mid x)$
the KL term is	a penalty we add, to keep clouds near $N(0, I)$	part of the ELBO: it comes out of Bayes' rule
the squared error is	reconstruction error	$-\log p_\theta(x \mid z)$: Gaussian pixels of size σ
its weight $1/2\sigma^2$	a knob to tune (β, section 6.6)	set by the pixel noise σ
to generate	$z \sim N(0, I)$, then decode	the same: sample the prior, follow the arrow

Takeaway. Same network, same loss. Road 1 says what the VAE does; Road 2 says why that loss is the right one.

Road 1 is the quick intuition; Road 2 is the reason. Both give the network we now train.

▶ Notebook 6.4

The loss for one image is the negative ELBO

$$L(\phi, \theta; x) = \underbrace{\mathbb{E}_{q_\phi(z|x)}\left[\frac{1}{2\sigma^2}\|x - f_\theta(z)\|^2\right]}_{\text{reconstruction}} + \underbrace{\mathrm{KL}\big(q_\phi(z \mid x) \,\|\, \mathcal{N}(0, I)\big)}_{\text{stay near the prior}}$$

$$\text{reconstruction} \approx \frac{1}{2\sigma^2}\|x - f_\theta(\mu + \sigma \odot \epsilon)\|^2, \qquad \epsilon \sim \mathcal{N}(0, I) \text{ (one draw)}$$

$$\mathrm{KL} = \frac{1}{2}\sum_{k=1}^{2}\left(\mu_k^2 + \sigma_k^2 - \log \sigma_k^2 - 1\right) \qquad \text{(Lecture 1, closed-form KL)}$$

↗ Lecture 1: Monte Carlo estimates | ↗ Lecture 1: reparameterisation | ↗ Lecture 1: diagonal-Gaussian KL

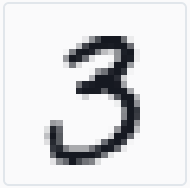
> The KL between two diagonal Gaussians has a closed form: no sampling needed.

▶ Notebook 6.4

One step for our 3, with numbers

$$\mu = (-0.211,\ 0.070), \quad \sigma = (0.013,\ 0.010), \quad \epsilon = (0.5,\ -1.0)$$

$$z = \mu + \sigma \odot \epsilon = (-0.204,\ 0.060)$$



vs



$$\frac{1}{2(0.2)^2} \times 23.18 = 289.8$$

$$\mathrm{KL} = 7.94 \qquad L = 289.8 + 7.94 = 297.7$$

Check: L − 541.4 = −243.7, close to −ELBO = −243.4. The loss drops the constant, and one draw is noisy.

> Add the KL to the prior: the loss for this image on this step.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 6.4

The VAE in PyTorch: three changes to the autoencoder


```python
class VAE(AutoEncoder):              # same decoder
    def __init__(self, z=2):
        super().__init__(z)
        self.encoder[-1] = nn.Linear(400, 2 * z)  # (1)

    def forward(self, x):
        mu, logvar = self.encoder(x).chunk(2, dim=1)
        eps = torch.randn_like(mu)
        z = mu + torch.exp(0.5 * logvar) * eps       # (2)
        return self.decoder(z), mu, logvar

x_hat, mu, logvar = model(x)
recon = ((x_hat - x)**2).sum(1) / (2 * 0.2**2)
kl = 0.5 * (mu**2 + logvar.exp() - logvar - 1).sum(1)  # (3)
loss = (recon + kl).mean()
```

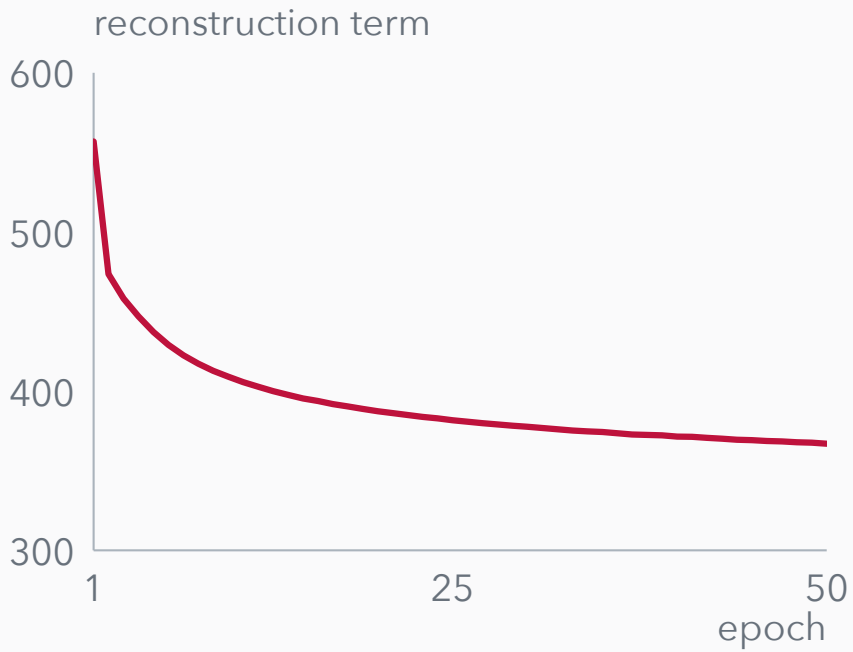
(1) the encoder outputs 2z numbers, (2) the code is a reparameterised draw, (3) the loss adds the KL.

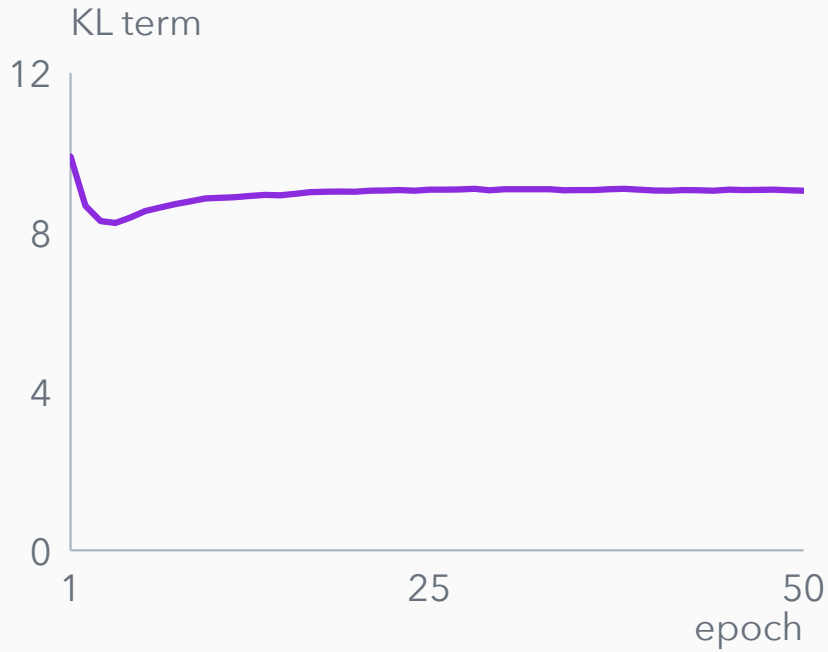
Training loop: identical to the autoencoder's.

> Everything else, the batches, Adam and 50 passes over the data, is the autoencoder's loop.

▶ Notebook 6.4

While training: reconstruction falls, the KL settles





After 50 epochs: reconstruction 367, KL 9.0 per image.

> The KL stays near 9 nats per image: each code is allowed to say about 9 nats more than the prior, no more.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 6.5

Now sample: z ~ N(0, I), then decode

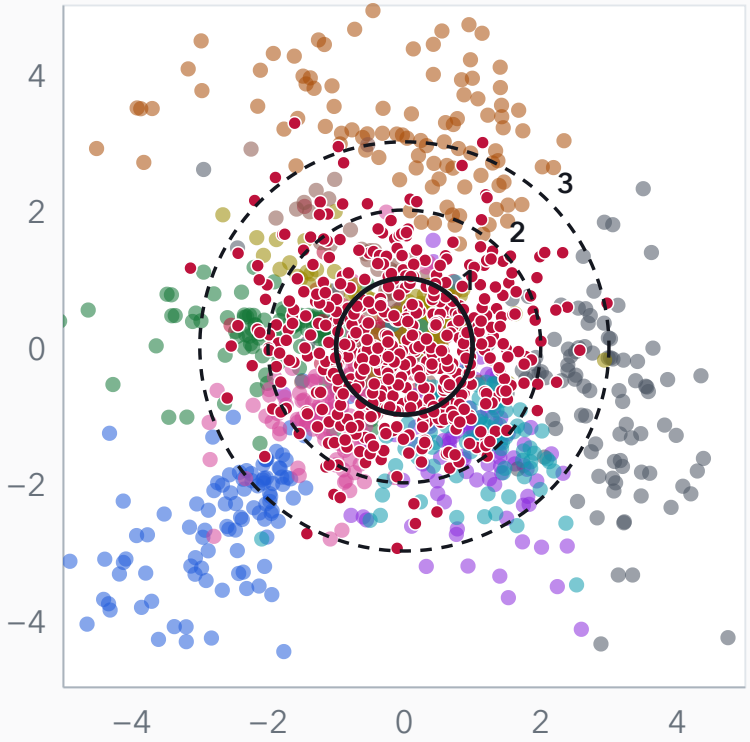


> Through the 16-number VAE: sharper and more varied. These include the fakes from the start of the lecture.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 6.5

The codes now fill N(0, I)



Codes of all 10,000 test digits: mean (−0.2, −0.2), spread (1.8, 1.7).

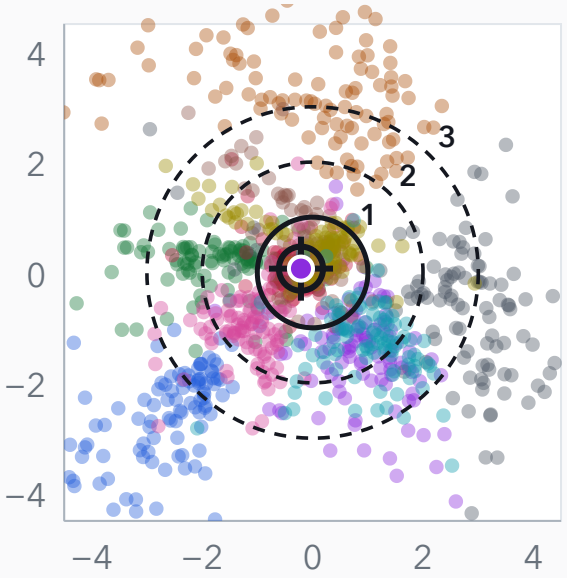
Random draws (red) now land among real codes, on every digit.

> Overlay 500 draws from N(0, I): they fall on real codes of every digit. That is why sampling works.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 6.5

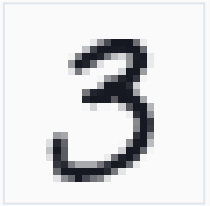
Your turn again: the same three ways, now with the VAE



0 1 2 3 4 5 6 7 8 9

decoder of the VAE, run in your browser





the digit we encoded

circles: N(0, I)
at radius 1, 2, 3

z = (−0.21, 0.07)
the code of our 3, decoded

> Random codes, encoded test digits, or click anywhere. Switch between the VAE and the autoencoder to compare.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 6.5

Decode a grid of codes: digits everywhere

VAE: codes from −3 to 3 in each direction



VAE: walk from the same 1 to the same 0



16-number VAE



Walk from the same 1 to the same 0: every step is a plausible digit.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 6.6

Put a weight on the KL: β

$$L_\beta = \mathbb{E}_q\left[\frac{1}{2\sigma^2}\|x - f_\theta(z)\|^2\right] + \beta\,\mathrm{KL}\big(q(z \mid x)\,\|\,\mathcal{N}(0, I)\big)$$

β = 1 is the true ELBO. Changing β is the same as assuming noisier or cleaner pixels (a different σ).

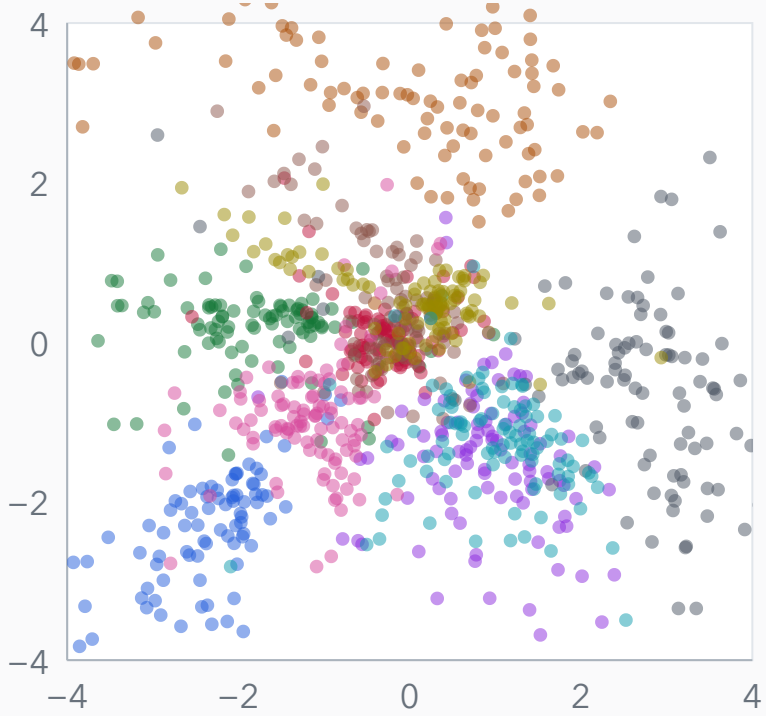
Small β
codes free to spread: close to an autoencoder

Large β
codes squeezed towards the prior: closer to N(0, I), less detail

> Predict before we look: what happens at β = 0, and at very large β?

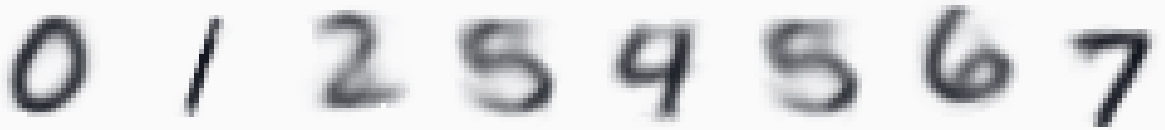
▶ Notebook 6.6

Seven VAEs, β from 0 to 256: try them

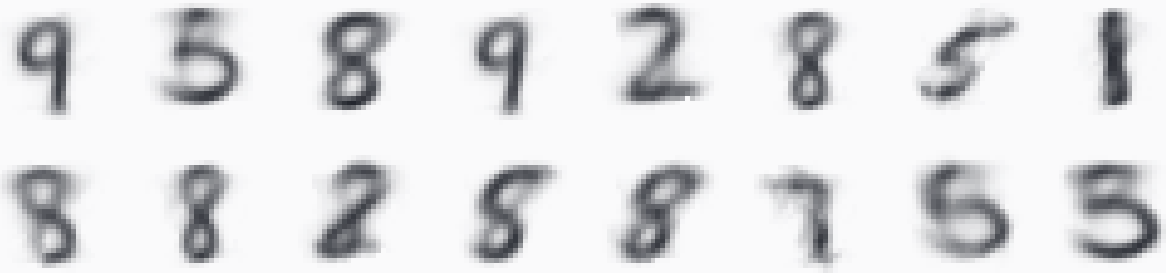


β = 1 KL 9.0 · squared error 30.8 · average σ of q 0.034

reconstructions of 8 test digits



16 samples: z ~ N(0, I), then decode



distance between two 0s ÷ distance from a 0 to a 1: 0.23

> Click through β. Watch three things: how spread the codes are, the reconstructions, and the samples.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 6.6

β = 0: is it just an autoencoder? (An exam question)

β	KL per image	average σ of q	spread of codes
0	68.3	0.0001	5.36
1	9.0	0.0343	1.68
256	0.0	1.0001	0.00

With β = 0 nothing stops q from shrinking.

Structurally it still draws $z = \mu + \sigma\varepsilon$, unlike an autoencoder. But training drives σ to 0.0001, so in practice it behaves like one: spread-out codes, poor samples.

> But training drives σ almost to zero, and then it behaves like one: spread-out codes and poor samples.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 6.6

β = 256: every code decodes to the same blur

β = 256, recon.



β = 256, samples



KL = 0.00: q(z | x) equals the prior for every image.

The decoder learns to ignore z and outputs the average digit. This is called **posterior collapse**.

> Then z carries no information, so the decoder ignores it and draws the average digit every time.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

▶ Notebook 6.7

The whole lecture in one table

model	how it generates	how it is fitted	what goes wrong
Gaussian or mixture (Part 2)	pick a bump, draw a number	maximum likelihood	fine for a few numbers
Gaussians on PCA numbers (3)	sample a code, decode linearly	maximum likelihood, then EM	straight-line decoder: blurry
autoencoder (4–5)	no rule for choosing codes	squared error	random codes miss the data
autoencoder + mixture (5)	sample a fitted code, decode	two separate fits	no single p(x)
VAE (6)	$z \sim N(0, I)$, decode	the ELBO: reconstruction + KL	soft samples; collapse if β is large

> From one number per example to 784, from formulas to networks: always a hidden code first, then the data.

▶ Notebook 6.7

Next: can we drop the likelihood altogether?

VAE samples are soft because each pixel is an independent Gaussian around a mean image.

Next: a generator trained by a second network that judges real against fake (GANs). Then: many small denoising steps (diffusion).

> Next lectures replace it: a learned critic (GANs) and repeated denoising (diffusion).

A sparse autoencoder: a KL between two coins

Wide hidden layer, but each unit should be on only rarely, say for ρ = 5% of images.

$$L = \text{reconstruction} + \lambda \sum_j \mathrm{KL}\big(\text{Bernoulli}(\rho) \,\|\, \text{Bernoulli}(\hat{\rho}_j)\big)$$

$\hat{\rho}_j$ is unit j's average activation over the data. Lecture 1's Bernoulli KL, used as a penalty.

> The penalty is the KL between two coins: the target rate ρ and the unit's actual rate.

Ng, "Sparse autoencoder", CS294A lecture notes; Lilian Weng, "From Autoencoder to Beta-VAE".

Held-out scores of the PCA mixtures

d	K = 1	K = 5	K = 10	K = 20
2	−4.4	−4.1	−4.1	−4.1
10	−18.5	−16.0	−15.3	−14.8
20	−31.6	−25.6	−23.9	−21.9
50	−55.9	−40.4	−32.1	−26.0

Average log-likelihood of the 10,000 test codes. Compare along a row only: different d are densities in different spaces.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

EM in one slide: the classic way to fit a mixture

$$r_{ik} = \frac{\pi_k \mathcal{N}(x_i; \mu_k, \Sigma_k)}{\sum_j \pi_j \mathcal{N}(x_i; \mu_j, \Sigma_j)} \qquad \text{(E: who made each point?)}$$

$$\pi_k = \tfrac{1}{n}\textstyle\sum_i r_{ik}, \quad \mu_k = \frac{\sum_i r_{ik} x_i}{\sum_i r_{ik}}, \quad \Sigma_k = \frac{\sum_i r_{ik}(x_i - \mu_k)(x_i - \mu_k)^\top}{\sum_i r_{ik}} \qquad \text{(M)}$$

> M step: refit each bump to the points, weighted by those probabilities. Repeat: the likelihood never decreases.

Second try: one 784-dimensional Gaussian with a full covariance

$$p(x) = \mathcal{N}(x;\, m, \Sigma), \qquad \Sigma \in \mathbb{R}^{784 \times 784}$$

308,504 numbers: 784 for the mean, 307,720 for the covariance.

Samples have smoother, stroke-like blobs, but they are ghosts, not digits.

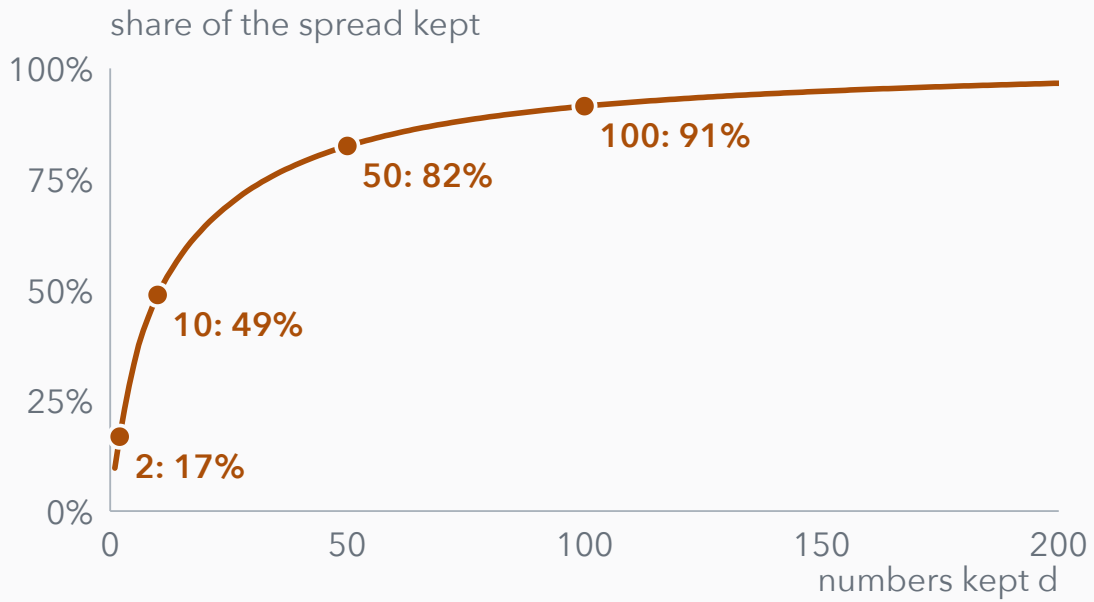
8 samples



> Samples are smoother, because neighbouring pixels now move together. Still not digits: one bump for ten kinds of digit, and far too many numbers.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

How many numbers do we need?



d = 10 keeps 49% of the spread; d = 50 keeps 82%.

784 numbers per digit, but about 50 carry most of the information.

> Digits really live near a much lower-dimensional surface. That is why compressing first is a good idea.

One Gaussian over d PCA numbers



> Over 50 numbers: more detail, but many samples are mixtures of two digits. One bump for ten kinds of digit.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

Search by code: the nearest digits to a query



Encode the query, encode all 60,000 training digits, return the 5 nearest codes.

Music apps and “vector databases” do the same with codes of songs, images and text.

> The nearest codes are other 1s with the same slant: the code keeps what matters about the digit.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

Denoising: damage the input, ask for the clean image



Error against the clean digits: plain 34.2, denoising 11.9.

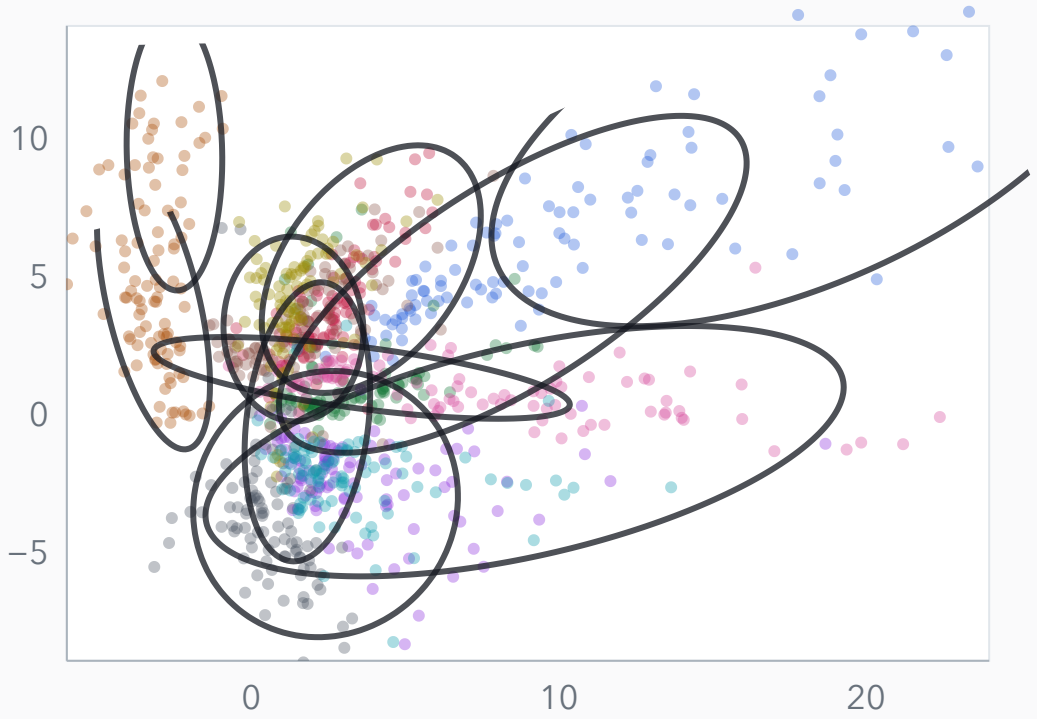
$$L = \frac{1}{n} \sum_i \left\| x_i - f_\theta(g_\phi(\tilde{x}_i)) \right\|^2$$

Input: the damaged $\tilde{x}$. Target: the clean x.

> Its loss compares the output with the clean image, so it must learn what digits look like, not copy its input.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

A fix we already know: fit a mixture to where the codes are



Part 3's recipe: fit 10 Gaussians to the codes of the 60,000 training digits, sample a code, decode.

2-number autoencoder + mixture



16-number autoencoder + mixture



The same with 16-number codes: sharper digits. A density fitted after training works.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

With 2 numbers we can add up rectangles; with 16 we cannot

$$\log p(x) \approx \log \sum_{\text{cells}} p(x \mid z_c)\, p(z_c)\, \Delta z = 244.07$$

Checked by importance sampling with 100,000 draws: 244.07.

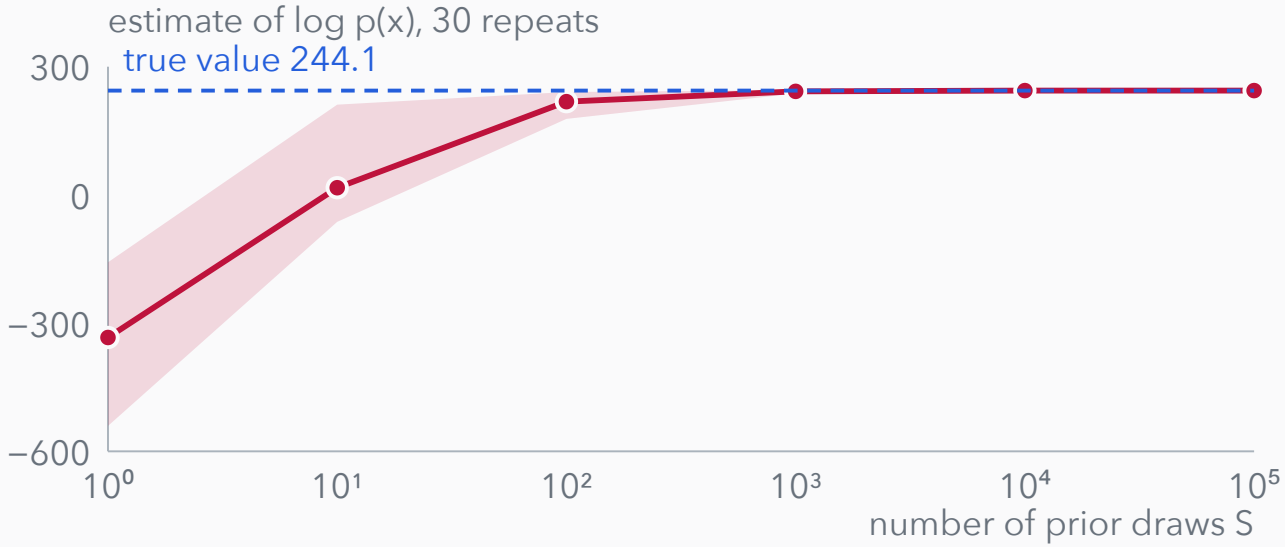
A grid with 161 points per axis (from −4 to 4 in steps of 0.05) has $161^2 \approx 26{,}000$ cells for 2 numbers, and $161^{16} \approx 10^{35}$ cells for 16. Lecture 1's intractable evidence, again.

> Grids grow exponentially with the number of code dimensions. We need another way.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

Average p(x | z) over random codes? Most draws are wasted

$$p(x) = \mathbb{E}_{z \sim p(z)}\left[p(x \mid z)\right] \approx \frac{1}{S}\sum_{s=1}^{S} p(x \mid z_s)$$



With 100 draws the estimate is still off by about 26.

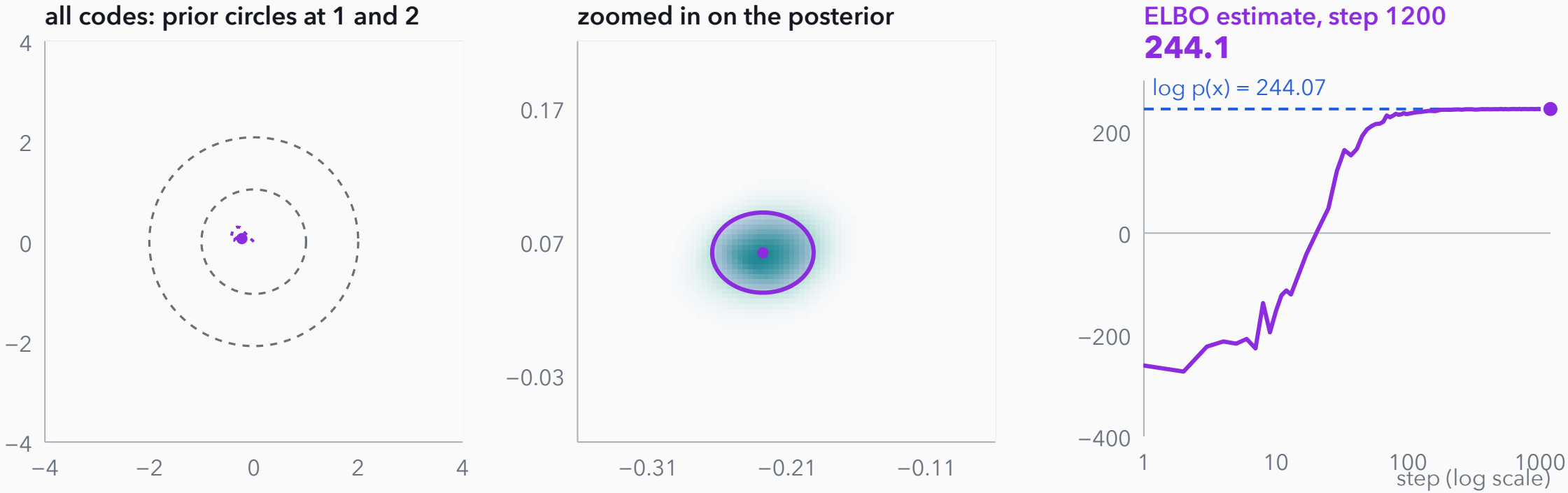
Of 100,000 prior draws, the best 10 carry 38% of the total.

Almost every code from the prior explains this 3 terribly. We need codes from the posterior's tiny patch.

> A handful of lucky draws carry the whole sum. We should draw codes where the posterior is, not from the prior.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

Fit q for our 3: press play



> q moves into the posterior's patch and shrinks to fit it. The ELBO ends at 244.05, just below log p(x) = 244.07.

Computed for this lecture: `scripts/science.py` from the networks trained by `scripts/train.py` (MNIST, fixed seeds).

60,000 images would mean 60,000 separate fits

Each image has its own posterior, so its own m and s: 1,200 steps each, and again for every new image.

Instead, train a network that reads an image and outputs its m and s in one pass: the **encoder**.

$$q_\phi(z \mid x) = \mathcal{N}\left(\mu_\phi(x),\ \operatorname{diag}\sigma_\phi(x)^2\right)$$

> One network with weights φ, shared by all images. This is called amortised inference: pay once in training, then every image is one forward pass.