

Variational Autoencoders: From Compression to Generation

Lecture Notes Consolidated with Claude AI — CS6140 Machine Learning

1 Where We Are: Three Things You Already Have

1.1 The autoencoder of HW5 Problem 1

HW5 Problem 1 builds, from the Week 9 layers, a network $8 \rightarrow 3 \rightarrow 8$ trained to output its own input. Split it in the middle and name the halves:

$$\underbrace{\mathbf{z} = f_{\phi}(\mathbf{x})}_{\text{encoder: Linear + sigmoid}} \qquad \underbrace{\hat{\mathbf{x}} = g_{\theta}(\mathbf{z})}_{\text{decoder: Linear + sigmoid}} \qquad \text{loss } \|\mathbf{x} - \hat{\mathbf{x}}\|^2.$$

ϕ and θ are just the weights of the first and second layer. Training is ordinary backprop. Because 3 hidden units must carry 8 distinct inputs, the network is forced to invent a *code*: rounded to 0/1, the hidden layer is a 3-bit binary code for the 8 inputs. The code $\mathbf{z}$ is called a **latent** representation (“latent” = hidden, never observed directly).

1.2 PCA is a linear autoencoder (HW3)

PCA with k components encodes $\mathbf{z} = W^{\top}(\mathbf{x} - \mathbf{m})$ and decodes $\hat{\mathbf{x}} = \mathbf{m} + W\mathbf{z}$, where the columns of W are the top- k principal directions. Among all *linear* encoder/decoder pairs with a k -dimensional code, this pair minimizes the squared reconstruction error. (A linear autoencoder trained by gradient descent on squared error learns the same k -dimensional subspace, though not necessarily the same basis inside it.) So an autoencoder is “nonlinear PCA”: same goal, a more flexible encoder and decoder.

1.3 A generative model with a latent variable (HW4)

HW4’s Gaussian mixture is a recipe for *producing* data:

1. pick a component $z \in \{1, \dots, K\}$ with probability π_z (the **prior** $p(z)$);
2. draw $\mathbf{x} \sim \mathcal{N}(\mu_z, \Sigma_z)$ (the **likelihood** $p(\mathbf{x} | z)$).

The E-step computes, for each data point, the **posterior** $p(z | \mathbf{x})$, i.e. the responsibilities

$$r_{ik} = \frac{\pi_k \mathcal{N}(\mathbf{x}_i; \mu_k, \Sigma_k)}{\sum_j \pi_j \mathcal{N}(\mathbf{x}_i; \mu_j, \Sigma_j)}.$$

Read this as “which latent value probably produced this $\mathbf{x}$?” That question is exactly what an *encoder* answers.

1.4 What an autoencoder cannot do

A **generative model** lets you draw *new* $\mathbf{x}$'s that look like the data. The GMM can (steps 1–2 above). The autoencoder cannot, for two reasons:

1. **No distribution over codes.** To generate, you would pick a $\mathbf{z}$ and decode it. But which $\mathbf{z}$? Training only cared about reconstructing, so the codes sit wherever reconstruction happened to put them: any scale, any location, clusters with gaps in between (Figure 3, left: codes spread over roughly ± 30).
2. **No reason for in-between codes to mean anything.** A $\mathbf{z}$ between two clusters was never produced by any training image, so nothing told the decoder what to do with it.

Key point: The VAE keeps the autoencoder's two networks and adds one thing: it *decides in advance* where the codes should live, a standard Gaussian $\mathcal{N}(0, I)$, and penalizes the encoder for putting them elsewhere. Then “generate” has an obvious meaning: draw $\mathbf{z} \sim \mathcal{N}(0, I)$ and decode.

2 The Big Picture: an Autoencoder Whose Code Is Random

Before any probability, here is the network (Figure 1). Compare it with the 8-3-8 autoencoder of §1: the shape is the same. An encoder squeezes the image $\mathbf{x}$ into a small code $\mathbf{z}$ (here just 2 numbers), and a decoder rebuilds the image from the code. Everything in gray, blue and green is a Week 9 fully-connected layer. Only two things are new:

1. The encoder does not output a code. It outputs, for each code dimension, a *mean* μ and a (log) *variance* $\log \sigma^2$: “this image’s code is probably near μ , give or take σ .”
2. The code is then *drawn at random* from that range, $\mathbf{z} = \mu + \sigma \odot \epsilon$, with fresh noise ϵ each time. This step has no weights.

Training adds one extra loss term that keeps every image’s range close to $\mathcal{N}(0, I)$ (the key point above). After training the encoder is not needed at all: draw $\mathbf{z} \sim \mathcal{N}(0, I)$ and decode.

The rest of the note explains why this design works: why a random code is the right idea (§3); how to measure “close to $\mathcal{N}(0, I)$ ” (§4); where the loss comes from (§5); the whole training pipeline, box by box (§6, Figure 2); how gradients get through a random step (§7); and what the extra loss term buys in practice (§8).

3 A Continuous Latent Variable

Replace the GMM’s discrete z by a continuous one:

$$\mathbf{z} \sim p(\mathbf{z}) = \mathcal{N}(0, I), \quad \mathbf{x} \sim p_\theta(\mathbf{x} | \mathbf{z}) \quad (\text{the decoder turns } \mathbf{z} \text{ into a distribution over } \mathbf{x}).$$

Maximum likelihood (Weeks 1–4) would maximize $\sum_i \log p_\theta(\mathbf{x}_i)$, where

$$p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{x} | \mathbf{z}) p(\mathbf{z}) d\mathbf{z}, \tag{1}$$

the GMM’s $\sum_k \pi_k \mathcal{N}(\mathbf{x}; \mu_k, \Sigma_k)$ with the sum over components becoming an integral over codes.

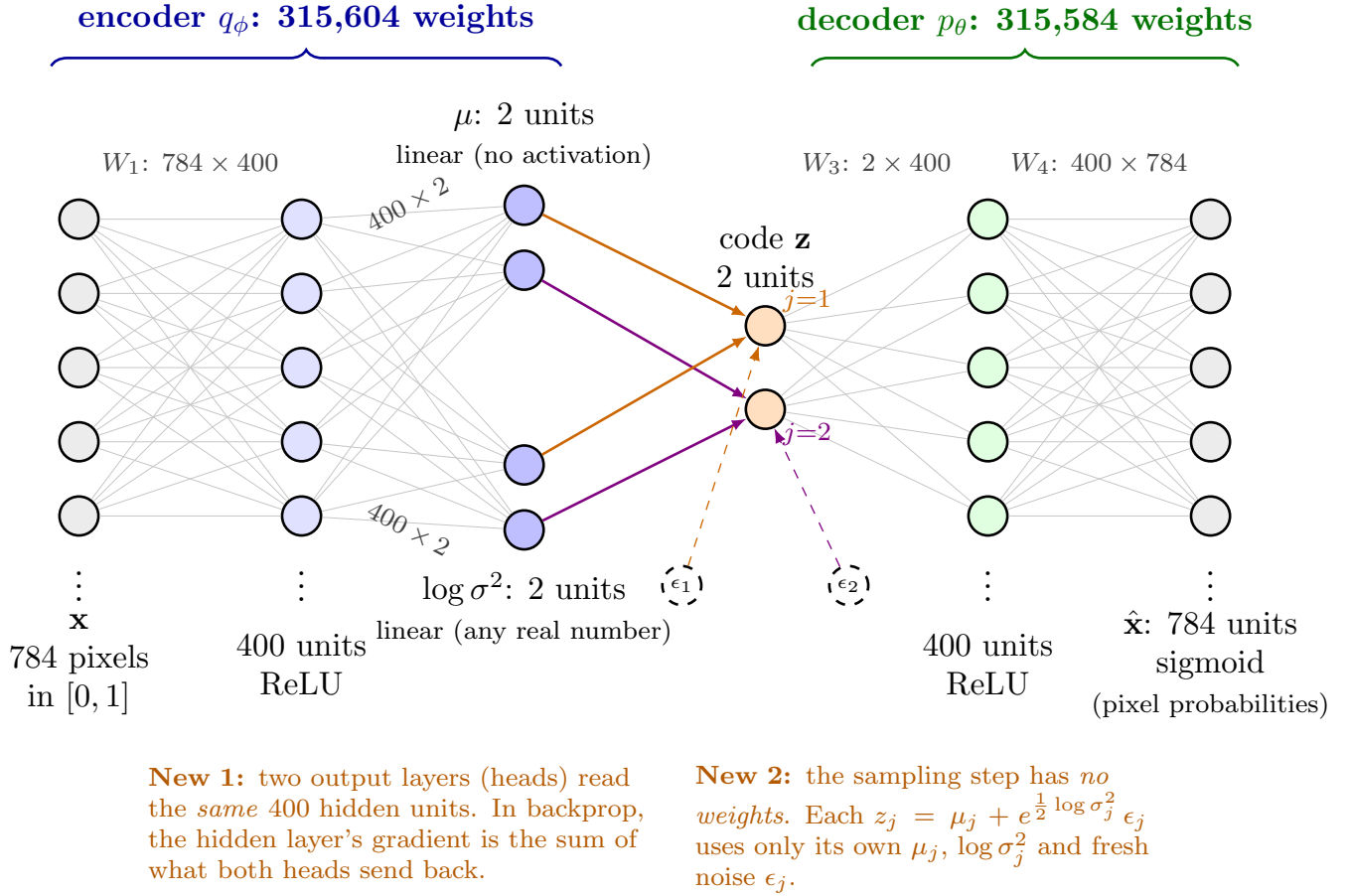


Figure 1: The VAE's layers (HW5 sizes; only a few units of each layer are drawn). Lines between columns are ordinary dense connections with weights. The colored arrows into the code $\mathbf{z}$ are *not* weights: each code unit is computed from its own μ_j , $\log \sigma_j^2$ and noise ϵ_j (one color per latent dimension). About 631,000 weights in total, almost all in the first and last layer.

Worked example: a linear decoder, where everything is computable Take a 1-D code and 2-D data, with a *linear* decoder plus Gaussian noise:

$$z \sim \mathcal{N}(0, 1), \quad \mathbf{x} = \mathbf{w}z + \boldsymbol{\varepsilon}, \quad \boldsymbol{\varepsilon} \sim \mathcal{N}(0, s^2 I), \quad \mathbf{w} = (2, 1), \quad s^2 = 1.$$

This model is called **probabilistic PCA**. Everything is Gaussian, so we can compute:

- **The marginal** (Eq. 1): $\mathbf{x}$ is a linear function of Gaussians, hence Gaussian, with mean 0 and covariance $\mathbf{w}\mathbf{w}^\top + s^2 I = \begin{pmatrix} 5 & 2 \\ 2 & 2 \end{pmatrix}$. The likelihood is available in closed form.
- **The posterior** (Bayes' rule; complete the square in z): with $M = \mathbf{w}^\top \mathbf{w} + s^2 = 6$,

$$p(z | \mathbf{x}) = \mathcal{N}\left(\frac{\mathbf{w}^\top \mathbf{x}}{M}, \frac{s^2}{M}\right).$$

For $\mathbf{x} = (3, 1)$: mean = $(2 \cdot 3 + 1 \cdot 1)/6 = 7/6 \approx 1.167$, variance = $1/6$. (The demo checks this by brute force on a grid.)

Two things to notice. First, the exact “encoder” $\mathbf{x} \mapsto (\text{posterior mean}, \text{posterior variance})$ is a **linear layer**: mean = $(\frac{1}{3}, \frac{1}{6}) \cdot \mathbf{x}$, plus a constant variance. Second, as the noise $s^2 \rightarrow 0$, the mean becomes $\mathbf{w}^\top \mathbf{x} / \|\mathbf{w}\|^2 = 7/5 = 1.4$, the projection of $\mathbf{x}$ onto the direction $\mathbf{w}$: PCA's encoder from HW3.

Now make the decoder a neural network. Replace $\mathbf{w}z$ by $g_\theta(\mathbf{z})$, a Week 9 MLP. Then:

- the integral in Eq. (1) has no closed form (there is no formula for “integrate a neural network against a Gaussian”); and
- the posterior $p_\theta(\mathbf{z} \mid \mathbf{x}) \propto p_\theta(\mathbf{x} \mid \mathbf{z})p(\mathbf{z})$ is no longer Gaussian and has no formula either. You cannot “invert” a multi-layer network by completing a square.

In the GMM, the posterior was a normalized list of K numbers, cheap to compute exactly. Here it is a whole unknown distribution over $\mathbb{R}^d$, different for every $\mathbf{x}$.

Remark. **Why not just run EM longer?** EM’s loop is “E-step, M-step, repeat”, and more iterations help only if each E-step can be carried out. The obstacle here is not how many iterations we run but that a *single* E-step is impossible: it asks for $p_\theta(\mathbf{z} \mid \mathbf{x})$, which needs the normalizing integral $\int p_\theta(\mathbf{x} \mid \mathbf{z})p(\mathbf{z}) d\mathbf{z}$ over all of $\mathbb{R}^d$, and a neural network gives no formula for it. Repeating a step you cannot compute does not help. There is also a cost argument. Even an *approximate* per-point E-step (say, a small optimization over q for each image) would have to be redone for all 60,000 training images at every iteration, and again for every new image at test time. The VAE’s answer, in §6, replaces all of those separate posteriors by one network that has learned to output a good guess for any $\mathbf{x}$ in one forward pass.

Key point: The plan: (1) find a quantity we *can* compute that is never larger than $\log p_\theta(\mathbf{x})$, and maximize it instead; (2) since the exact posterior is out of reach, let a *second* network guess it, for each $\mathbf{x}$. In the linear example that guess can be exact (a linear layer). With a neural decoder it can only be approximate. To say how good a guess is, we need a way to compare two distributions.

4 Comparing Two Distributions: KL Divergence

4.1 Definition, with a discrete example

For two distributions p, q over the same outcomes,

$$\text{KL}(p \parallel q) = \sum_x p(x) \log \frac{p(x)}{q(x)} \quad \left(\text{continuous: } \int p(x) \log \frac{p(x)}{q(x)} dx \right).$$

Read it as “the average extra surprise, $\log \frac{1}{q} - \log \frac{1}{p}$, you pay for believing q when outcomes really come from p ”.

Worked example: three outcomes $p = (\frac{1}{2}, \frac{1}{4}, \frac{1}{4})$, $q = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$:

$$\text{KL}(p \parallel q) = \frac{1}{2} \ln \frac{1/2}{1/3} + 2 \cdot \frac{1}{4} \ln \frac{1/4}{1/3} = \frac{1}{2}(0.4055) + \frac{1}{2}(-0.2877) = 0.0589,$$

$$\text{KL}(q \parallel p) = \frac{1}{3} \left(\ln \frac{1/3}{1/2} + 2 \ln \frac{1/3}{1/4} \right) = \frac{1}{3}(-0.4055 + 0.5754) = 0.0566.$$

Properties to remember (all visible in the example):

- $\text{KL}(p \parallel q) \geq 0$, with equality exactly when $p = q$ (individual terms can be negative; the sum cannot).
- It is *not symmetric* ($0.0589 \neq 0.0566$), so it is not a distance. Which argument comes first matters, and the VAE uses a specific order.
- If $q(x) = 0$ where $p(x) > 0$, the KL is infinite: q is “infinitely surprised”.

4.2 You have been minimizing KL all along

The **cross-entropy** $H(p, q) = -\sum_x p(x) \log q(x)$ splits as

$$H(p, q) = \underbrace{-\sum_x p(x) \log p(x)}_{\text{entropy } H(p), \text{ fixed}} + \text{KL}(p \parallel q). \quad \text{Example: } 1.0986 = 1.0397 + 0.0589.$$

Week 9's softmax/cross-entropy loss is $H(p_{\text{data}}, q_{\text{model}})$ with p_{data} the one-hot label. Since $H(p)$ does not depend on the model, **minimizing cross-entropy = maximizing likelihood = minimizing $\text{KL}(p_{\text{data}} \parallel q_{\text{model}})$** . KL is not new machinery; it is the thing maximum likelihood was quietly minimizing since Week 1.

4.3 KL between two Gaussians

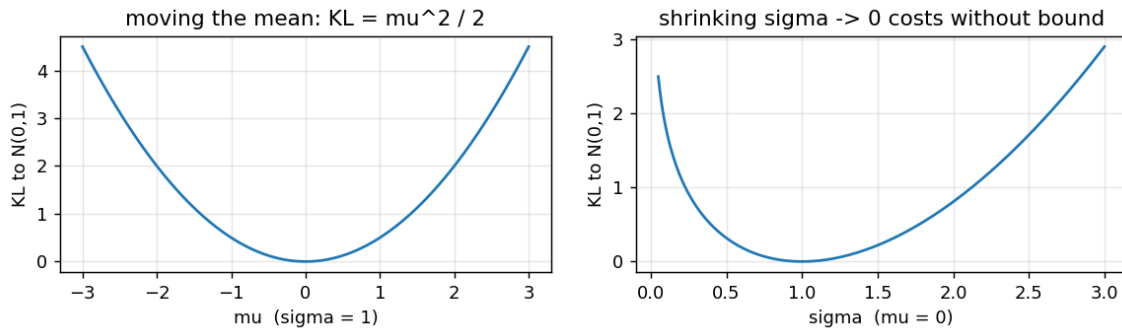
The VAE needs one specific KL: an encoder's Gaussian $q = \mathcal{N}(\mu, \sigma^2)$ against the prior $p = \mathcal{N}(0, 1)$. Three lines, using $\log \mathcal{N}(z; \mu, \sigma^2) = -\frac{1}{2} \log(2\pi\sigma^2) - \frac{(z-\mu)^2}{2\sigma^2}$, $\mathbb{E}_q[(z-\mu)^2] = \sigma^2$ and $\mathbb{E}_q[z^2] = \mu^2 + \sigma^2$:

$$\begin{aligned} \text{KL}(q \parallel p) &= \mathbb{E}_q[\log q(z) - \log p(z)] = \left(-\frac{1}{2} \log(2\pi\sigma^2) - \frac{1}{2} \right) - \left(-\frac{1}{2} \log(2\pi) - \frac{1}{2}(\mu^2 + \sigma^2) \right) \\ &= \frac{1}{2}(\mu^2 + \sigma^2 - 1 - \log \sigma^2). \end{aligned} \quad (2)$$

For a d -dimensional code with independent coordinates (a diagonal covariance), the KL is the sum over coordinates:

$$\text{KL}(\mathcal{N}(\mu, \text{diag } \sigma^2) \parallel \mathcal{N}(0, I)) = \frac{1}{2} \sum_{j=1}^d (\mu_j^2 + \sigma_j^2 - 1 - \log \sigma_j^2). \quad (3)$$

μ	σ	KL to $\mathcal{N}(0, 1)$	reading
0	1	0	q is the prior: free
1	1	0.5	moving the mean costs $\mu^2/2$
0	2	0.807	too wide costs
0	0.5	0.318	too narrow costs
0	0.1	1.808	nearly a point: cost grows like $-\log \sigma$, without bound



Key point: Read Eq. (2) as a *price list* the encoder pays for each code: move the code's mean away from 0 and you pay $\mu^2/2$; shrink its spread toward a point and you pay without bound. The second item is what stops a VAE from collapsing back into a plain autoencoder, whose codes are points ($\sigma = 0$).

5 The Evidence Lower Bound (ELBO)

5.1 One identity, true for any guess q

Let $q(\mathbf{z})$ be *any* distribution over codes, our guess of the posterior for one fixed $\mathbf{x}$. Since $p(\mathbf{x}) = p(\mathbf{x}, \mathbf{z})/p(\mathbf{z} | \mathbf{x})$ for every $\mathbf{z}$, we can take the average over $\mathbf{z} \sim q$ of both sides' logs and insert q on top and bottom:

$$\log p(\mathbf{x}) = \mathbb{E}_q \left[\log \frac{p(\mathbf{x}, \mathbf{z})}{p(\mathbf{z} | \mathbf{x})} \right] = \underbrace{\mathbb{E}_q \left[\log \frac{p(\mathbf{x}, \mathbf{z})}{q(\mathbf{z})} \right]}_{\text{ELBO}(q)} + \underbrace{\mathbb{E}_q \left[\log \frac{q(\mathbf{z})}{p(\mathbf{z} | \mathbf{x})} \right]}_{\text{KL}(q \| p(\mathbf{z} | \mathbf{x})) \geq 0}. \quad (4)$$

Because the KL is ≥ 0 , $\text{ELBO}(q) \leq \log p(\mathbf{x})$: the **evidence lower bound** (“evidence” is another name for $p(\mathbf{x})$). The gap is exactly how wrong the guess q is.

Worked example: a two-state latent variable (a GMM with $K = 2$, one data point) Prior $p(z) = (\frac{1}{2}, \frac{1}{2})$. For the observed point, $p(\mathbf{x} | z=1) = 0.2$ and $p(\mathbf{x} | z=2) = 0.6$. Then $p(\mathbf{x}) = \frac{1}{2}(0.2) + \frac{1}{2}(0.6) = 0.4$, $\log p(\mathbf{x}) = -0.916$, and the posterior (the responsibilities) is $(0.25, 0.75)$. Try three guesses:

guess $q(z)$	$\mathbb{E}_q[\log p(\mathbf{x} z)]$	$\text{KL}(q \ p(z))$	ELBO	gap	$\text{KL}(q \ \text{posterior})$
(0.5, 0.5)	-1.060	0	-1.060	0.144	0.144
(0.1, 0.9)	-0.621	0.368	-0.989	0.072	0.072
(0.25, 0.75) = posterior	-0.785	0.131	-0.916	0	0

Every row satisfies Eq. (4): $\text{ELBO} + \text{gap} = -0.916$. The better the guess, the higher the ELBO. The perfect guess makes the bound tight.

5.2 This is EM

EM's E-step sets q to the exact posterior (the responsibilities): the gap becomes 0 and the ELBO touches $\log p(\mathbf{x})$. The M-step then raises the ELBO over the model parameters. This is what HW4 Problem 3(C) asks you to identify: $q(z | \mathbf{x}) = r_{ik}$ and $p(z) = \pi_k$.

5.3 The form the VAE uses

Split $p(\mathbf{x}, \mathbf{z}) = p(\mathbf{x} | \mathbf{z})p(\mathbf{z})$ inside the ELBO:

$$\text{ELBO}(q) = \underbrace{\mathbb{E}_q[\log p(\mathbf{x} | \mathbf{z})]}_{\text{reconstruction: codes from } q \text{ should explain } \mathbf{x}} - \underbrace{\text{KL}(q \| p(\mathbf{z}))}_{\text{regularizer: stay close to the prior}}. \quad (5)$$

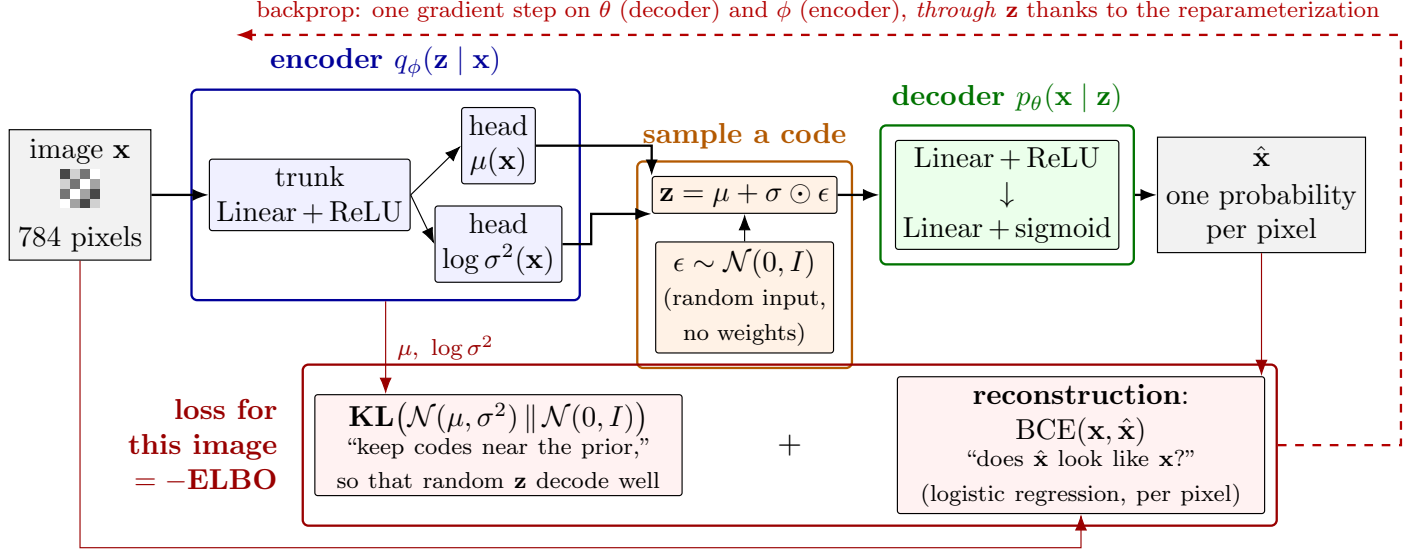
Check with the table's second row: $-0.621 - 0.368 = -0.989$. The two terms pull in opposite directions. The reconstruction term wants q concentrated on the code that explains $\mathbf{x}$ best ($z = 2$). The KL term wants q to look like the prior. The best compromise is the true posterior.

Key point: The VAE's loss is $-\text{ELBO}$ from Eq. (5), averaged over the data: a reconstruction error (the autoencoder's loss) plus the KL “price” of §4 (the new part). Maximizing it does two jobs at once: it improves the model (raises $\log p(\mathbf{x})$) and it improves the guess q (shrinks the gap).

6 From the ELBO to a Network

Figure 2 shows the whole method at a glance: what happens to one training image, which loss terms it produces and why, and how new images are generated afterwards. The rest of this section and the next explain each box.

TRAINING (one image at a time, repeated over the dataset)



GENERATION (after training): no encoder, no image in

This works because the KL term kept every training code inside the $\mathcal{N}(0, I)$ cloud, so a random code lands where the decoder was trained.

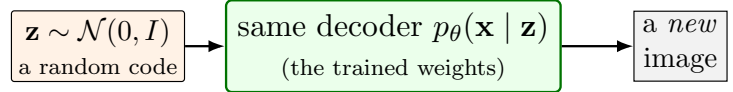


Figure 2: The VAE at a glance. **Training:** the encoder turns an image into a Gaussian guess $\mathcal{N}(\mu, \sigma^2)$ for its code; a code $\mathbf{z}$ is sampled from that guess (with the randomness moved into the input ϵ); the decoder turns $\mathbf{z}$ into pixel probabilities $\hat{\mathbf{x}}$. The loss has two terms with two jobs, and one backward pass updates both networks. **Generation:** throw the encoder away, draw $\mathbf{z}$ from the prior, decode.

6.1 Amortized inference: one network that guesses every posterior

EM computes a separate q (a responsibility vector) for every data point, exactly. For a VAE we choose the guess to be Gaussian with a diagonal covariance, and let a network produce its parameters from $\mathbf{x}$:

$$q_\phi(\mathbf{z} | \mathbf{x}) = \mathcal{N}(\mathbf{z}; \mu_\phi(\mathbf{x}), \text{diag} \sigma_\phi^2(\mathbf{x})).$$

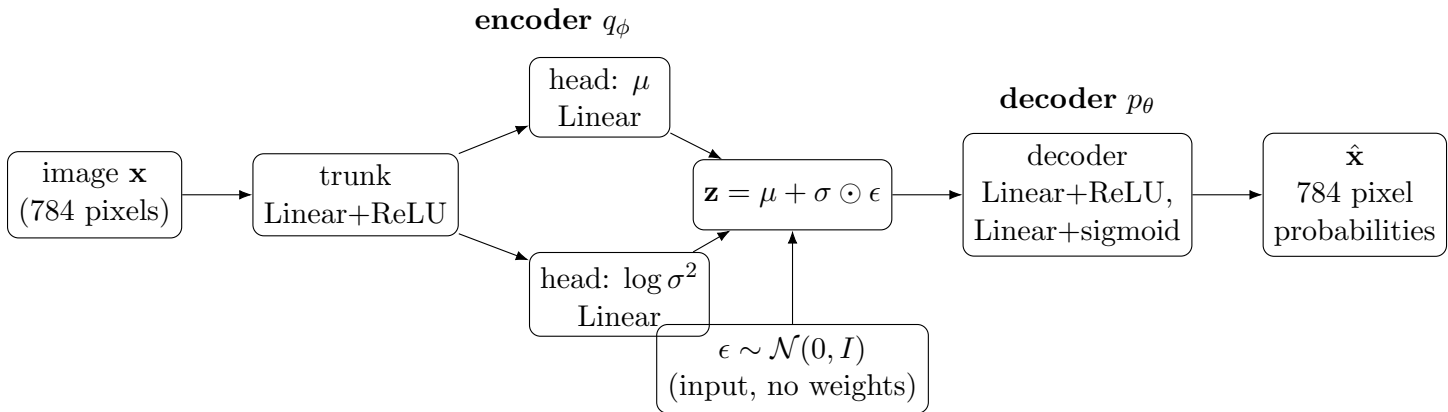
This network is the **encoder**. “Amortized” means the cost of learning to guess is paid once, in training, and then any new $\mathbf{x}$ gets its posterior guess from a single forward pass. In the linear example of §3, the best such network is exactly a linear layer. With a neural decoder we use an MLP and accept an approximation.

6.2 How new is this, structurally?

Less than it looks: every piece is a Week 9 layer.

- The **decoder** g_θ is the decoder half of HW5 Problem 1’s autoencoder, a stack of Linear + activation layers mapping a code to an image. The final sigmoid now outputs one probability per pixel.

- The **encoder** is a stack of Linear + activation layers (the *trunk*) followed by **two** output layers (*heads*) reading the same hidden vector: one outputs μ , one outputs $\log \sigma^2$. Using $\log \sigma^2$ lets the head output any real number while $\sigma^2 = e^{\log \sigma^2}$ stays positive.
- In backprop, the trunk's output feeds two layers, so its gradient is the *sum* of the gradients coming back from both heads (the Week 9 rule for a value used twice).



The layer-by-layer version of this diagram is Figure 1 in §2.

6.3 The reconstruction term is a Week 9 loss

$\log p_\theta(\mathbf{x} | \mathbf{z})$ depends on what we assume about pixels given the code:

- **Bernoulli pixels** (HW5, MNIST in $[0, 1]$): pixel j is “on” with probability $\hat{x}_j$ (the decoder’s sigmoid output), so $-\log p_\theta(\mathbf{x} | \mathbf{z}) = -\sum_j [x_j \log \hat{x}_j + (1 - x_j) \log(1 - \hat{x}_j)]$: the **binary cross-entropy**. This is the logistic-regression loss (Weeks 3–4) once per pixel, with all 784 “logistic regressions” sharing the decoder’s hidden layers.
- **Gaussian pixels** (the demo): $\mathbf{x} \sim \mathcal{N}(\hat{\mathbf{x}}, \sigma_x^2 I)$ gives $-\log p = \|\mathbf{x} - \hat{\mathbf{x}}\|^2 / (2\sigma_x^2) + \text{const}$: squared error, as in Week 1 (“least squares = Gaussian maximum likelihood”).

6.4 The loss for one image

$$\mathcal{L}(\mathbf{x}) = \underbrace{\text{BCE}(\mathbf{x}, g_\theta(\mathbf{z}))}_{\text{summed over pixels, } \mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})} + \underbrace{\frac{1}{2} \sum_j (\mu_j^2 + \sigma_j^2 - 1 - \log \sigma_j^2)}_{\text{Eq. (3), from the encoder's outputs}}. \quad (6)$$

The reconstruction expectation is estimated with **one** sampled $\mathbf{z}$ per image per step. That is noisy, but it is unbiased and averages out over many steps, just like SGD’s minibatch noise. The KL term needs no sampling at all.

Remark. Sum the BCE over pixels; don’t average it. Both terms are log-probabilities of one image, and averaging the reconstruction over 784 pixels quietly multiplies the KL’s weight by 784. The codes then collapse toward the prior and reconstructions go blurry.

7 Training Through a Random Node: the Reparameterization Trick

7.1 The problem

To train the encoder we need $\partial\mathcal{L}/\partial\phi$, and $\mathcal{L}$ contains “draw $\mathbf{z}$ from $\mathcal{N}(\mu, \sigma^2)$ ”. A draw is not a formula in μ and σ : if you nudge μ , the drawn number does not move smoothly. It is simply a different random number. Backprop has nothing to differentiate. (In PyTorch, `Normal(mu, sigma).sample()` returns a tensor with no gradient; the demo shows this.)

7.2 The trick: move the randomness into an input

From probability you already know how to standardize a Gaussian: $\epsilon = (z - \mu)/\sigma \sim \mathcal{N}(0, 1)$. Run it backwards:

$$\mathbf{z} = \mu_\phi(\mathbf{x}) + \sigma_\phi(\mathbf{x}) \odot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I). \quad (7)$$

Same distribution for $\mathbf{z}$, but now the randomness ϵ is an *input* that does not depend on ϕ , and $\mathbf{z}$ is an ordinary differentiable expression: $\partial z/\partial\mu = 1$ and $\partial z/\partial\sigma = \epsilon$. In the computation graph, ϵ is a leaf like the image $\mathbf{x}$, and backprop flows from the decoder, through $\mathbf{z}$, into both encoder heads.

Worked example: gradient of an expectation, by hand Let $f(z) = z^2$ and $z \sim \mathcal{N}(\mu, \sigma^2)$. We know $\mathbb{E}[f] = \mu^2 + \sigma^2$, so the true gradients are $\partial/\partial\mu = 2\mu$ and $\partial/\partial\sigma = 2\sigma$. With $z = \mu + \sigma\epsilon$:

$$\frac{\partial f}{\partial\mu} = 2(\mu + \sigma\epsilon), \quad \frac{\partial f}{\partial\sigma} = 2(\mu + \sigma\epsilon)\epsilon.$$

Take $\mu = 1$, $\sigma = \frac{1}{2}$ and two samples $\epsilon = -1, +1$:

ϵ	z	$\partial f/\partial\mu$	$\partial f/\partial\sigma$
-1	0.5	1	-1
+1	1.5	3	3
average		$2 = 2\mu \checkmark$	$1 = 2\sigma \checkmark$

(These two samples happen to be exact; with random ϵ 's the averages are right on average, e.g. 1.998 and 0.998 from 100,000 samples in the demo.)

Algorithm: VAE training (one minibatch)

1. Encoder forward: $\mu_\phi(\mathbf{x})$, $\log \sigma_\phi^2(\mathbf{x})$ for each image.
 2. Draw $\epsilon \sim \mathcal{N}(0, I)$; set $\mathbf{z} = \mu + \sigma \odot \epsilon$ with $\sigma = \exp(\frac{1}{2} \log \sigma^2)$.
 3. Decoder forward: $\hat{\mathbf{x}} = g_\theta(\mathbf{z})$.
 4. Loss (Eq. 6): BCE summed over pixels, plus the KL from step 1's outputs; average over the batch.
 5. Backprop through steps 3, 2, 1 and take one optimizer step on θ and ϕ together.
-

Key point: Compare with the GAN (next lecture): its generator also starts from noise, but the noise is only ever an input. There is no sampling step in the middle of the network, so it needs no trick. The VAE needs one because the encoder's *output* parameterizes a distribution that is then sampled.

8 What the KL Term Buys: Autoencoder vs. VAE (demo)

The demo trains the same network twice on sklearn's 8×8 digits with a 2-D code: once as a plain autoencoder (mean head only, reconstruction loss only), once as a VAE (sampling plus KL).

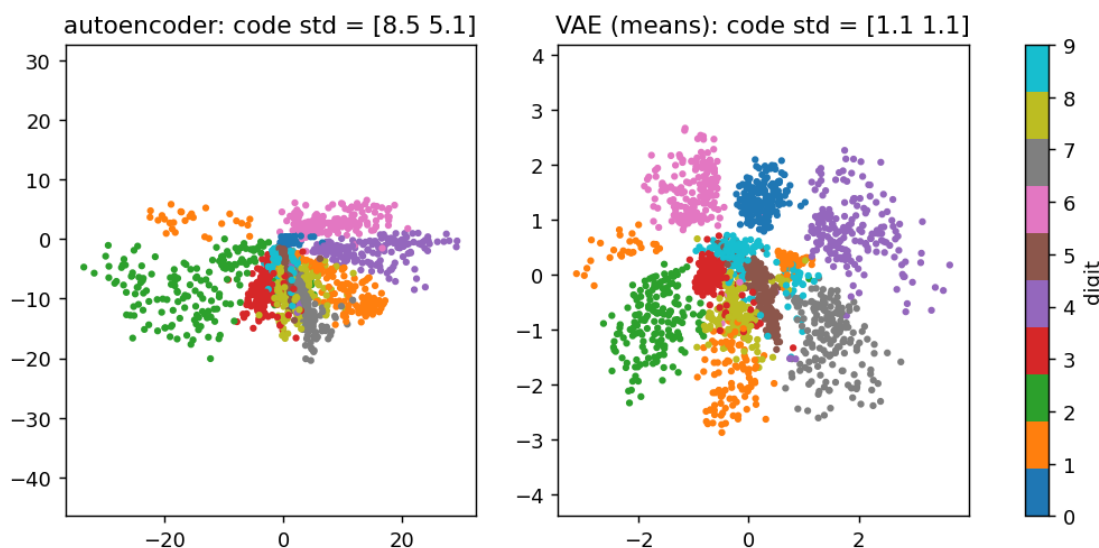
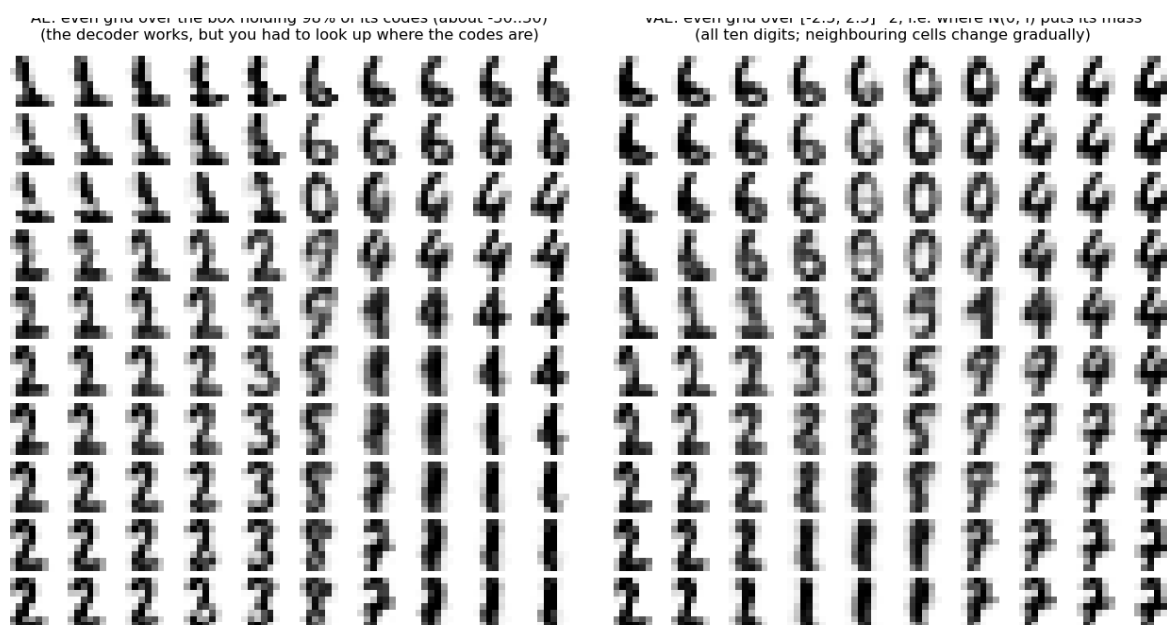


Figure 3: Codes of all 1797 images, colored by digit. Left: the autoencoder's codes, with arbitrary scale (standard deviation about 8 and 5) and uneven spread. Right: the VAE's means, pulled into the region where $\mathcal{N}(0, I)$ lives.



AE decoder on $\mathbf{z} \sim \mathcal{N}(0, I)$: these $\mathbf{z}$ cover only the centre of its code space (code std about 8), so only the few classes clustered near 0 appear



VAE decoder on the same $\mathbf{z}$: $\mathcal{N}(0, I)$ is where the KL term put the codes, so the samples spread over many classes (this is "generation")



What to take from it:

- With only 2 code dimensions and tiny images, *both* decoders make digit-like pictures somewhere. The plain autoencoder's decoder is not broken.
- The difference is **where the codes are**. For the autoencoder, you had to look up the code range after training, and $\mathbf{z} \sim \mathcal{N}(0, I)$ lands in a small central patch of its code space, so only the few classes that happen to sit near 0 come out. For the VAE, the KL term made $\mathcal{N}(0, I)$ the right place to sample, so the samples spread over many classes.
- The VAE's map is smooth: neighboring codes decode to similar images, which is why interpolating between two codes (HW5 Problem 3(B)) makes sense.
- Price: the VAE reconstructs slightly worse (its reconstruction term is 14.3 vs. 13.7). Every code must pay the KL price, and the codes are fuzzy ($\sigma > 0$), so the decoder must tolerate nearby codes.

Remark. **Posterior collapse** is the failure in the other direction. If the KL price is too high relative to reconstruction (e.g. the averaging mistake in §6), the cheapest solution is $\mu = 0, \sigma = 1$ for every image, which costs zero KL. The encoder then ignores its input and every reconstruction looks like the same average blur. Standard remedy: weight the KL by $\beta < 1$ early in training and raise it to 1 ("KL warm-up").

9 EM vs. VAE: the Same Idea, Two Implementations

	GMM fit by EM (HW4)	VAE (HW5)
latent $\mathbf{z}$	discrete, K values	continuous, $\mathbb{R}^d$
prior $p(\mathbf{z})$	π_k (learned)	$\mathcal{N}(0, I)$ (fixed)
decoder $p(\mathbf{x} \mathbf{z})$	Gaussian $\mathcal{N}(\mu_k, \Sigma_k)$	neural network g_θ
posterior guess	responsibilities r_{ik} , computed	encoder network, <i>learned</i> ,
$q(\mathbf{z} \mathbf{x})$	<i>exactly</i> by Bayes' rule	approximate
gap	0 after every E-step	> 0 , shrinks as the encoder
KL($q \parallel$ posterior)		improves
objective	ELBO (Eq. 5)	the same ELBO
optimization	alternate E-step / M-step	one gradient step on (θ, ϕ) together

EM's exactness comes from its small discrete latent space. The posterior is a list of K numbers, each one a density evaluation. With a continuous $\mathbf{z}$ and a neural decoder, the posterior is a distribution with no formula (§3), so neither the "exact list" nor the "complete the square" route exists. The VAE's answer is not a better E-step. It replaces the E-step by a function approximator trained on the same objective.

10 HW5 Problem 3: Map and Implementation Notes

- **(A)(i) the given architecture (two heads on one trunk):** §6.2. Why a 2-D code: so that (B) can *look* at the latent space directly. Reconstructions would be sharper with more dimensions, but that is not the point of this problem.
- **(A)(ii) reparameterization:** §7. The one-sentence explanation asked for is about what happens to the gradient at a sampling step.
- **(A)(iii) KL in closed form:** Eq. (3). The Monte-Carlo checkpoint averages $\log q(\mathbf{z}) - \log p(\mathbf{z})$ over samples $\mathbf{z} \sim q$ (the definition of KL). The demo’s cell 3 shows the same comparison for 1-D Gaussians.
- **(A)(iv) loss:** Eq. (6), BCE summed over pixels (see the remark in §6). Plot the two terms separately: reconstruction should fall steadily, and the KL typically rises first and then levels off, which is the encoder “buying” informative codes.
- **(B)(i)–(iii) latent space, sampling, interpolation:** §8. For the plain autoencoder comparison, expect differences in scale and location like the demo’s, not necessarily a broken decoder.
- **(C) written:** §3 (why no closed-form posterior) and §9.

11 Beyond HW5 (optional)

- **β -VAE:** weight the KL by β ; $\beta > 1$ trades reconstruction for more independent, often more interpretable code coordinates.
- **Conditional VAE:** feed the label y to encoder and decoder, so you can ask for “a 7”.
- **VQ-VAE:** a discrete codebook instead of a Gaussian code, used in several modern image and audio generators.
- **Diffusion models,** today’s standard image generators, are also trained with an ELBO, over a long chain of latent variables that gradually add and remove noise.

Take-home points

- An autoencoder compresses. A VAE compresses *and* fixes in advance where codes live ($\mathcal{N}(0, I)$), which turns the decoder into a generator.
- KL divergence compares distributions. Maximum likelihood is minimizing a KL, and the Gaussian-to- $\mathcal{N}(0, 1)$ KL is a simple “price list” (Eq. 2).
- $\log p(\mathbf{x}) = \text{ELBO}(q) + \text{KL}(q \parallel \text{posterior})$ for any guess q . EM makes the guess exact; a VAE learns it with an encoder network.
- The loss is reconstruction (BCE = per-pixel logistic regression) plus KL. The reparameterization $\mathbf{z} = \mu + \sigma\epsilon$ lets ordinary backprop train both networks at once.

References

- Kingma & Welling (2014), *Auto-Encoding Variational Bayes*, ICLR.
- Doersch (2016), *Tutorial on Variational Autoencoders*, arXiv:1606.05908.
- Tipping & Bishop (1999), *Probabilistic Principal Component Analysis*, JRSS-B (the linear example of §3).
- Murphy, *Probabilistic Machine Learning* (KMPP), the chapters on latent-variable models and VAEs.