Variational Autoencoders: Pre-reading Notes

We've spent the year building neural networks that take one kind of thing and turn it into another: text into translated text, images into captions. The pattern is always the same — an *encoder* compresses the input into some intermediate representation, and a *decoder* unpacks that representation into the output.

In this notebook we're going to do something genuinely new. We're going to build a network that can **generate** images: produce novel faces it has never seen before. That's a different kind of problem than translation or captioning. There's no input to translate from; we're conjuring outputs from nothing.

The trick, it turns out, is to look at our familiar encoder/decoder pattern from a strange angle. If we set things up just right, the decoder alone can be used as a generator. The challenge is figuring out what "just right" means. That's what the VAE is about.

Before we get there, let's review what we already know.

---

§1. Three encoders we've already built

We've encountered encoders in three previous units. They look different on the surface, but they're doing the same kind of thing.

The RNN encoder (NMT). We fed a source sentence one token at a time and read off the final hidden state. That hidden state was a single vector — say, 512 numbers — that "summarized" the entire sentence. The decoder then unrolled this vector back into a target-language sentence. Lossy compression: the source sentence has potentially hundreds of words, but the hidden state has only 512 numbers. Information had to be thrown away.

The Transformer encoder (NMT, captioning). Same idea but more refined: instead of one vector for the whole input, we produced a vector *per input token*. So a 20-word sentence became 20 contextualized vectors. Still lossy — each output vector is much smaller than "everything about that word in context" — but less aggressive than the RNN's single-vector bottleneck.

The ResNet encoder (captioning). For images, we cut a pretrained ResNet off before its classification head and used its spatial feature maps as the encoding. A 224×224 image got compressed into 49 feature vectors arranged in a 7×7 grid. Each vector summarizes a region of the original image.

The common theme: **an encoder is lossy compression into a representation a downstream task can use.** Different tasks like different shapes of representation (one vector, many vectors, a grid), but the underlying job is the same.

For the VAE, we're going to push compression even further. Each image will be encoded to *one* vector. For 64×64 face images, that's 12,288 pixel values being squeezed into 128 numbers — a roughly 100× compression. This is a much more aggressive bottleneck than the captioning encoder used.

Why so aggressive? It'll be clear in §3 once we look at why we're doing this in the first place.

---

§2. Autoencoders

Suppose someone hands you the problem "build a network that compresses images and decompresses them back" and asks you to design it from scratch. What would you do?

Almost certainly this: build an encoder (compress to a small vector), build a decoder (expand back to an image), train them together to minimize the difference between the input and the reconstruction.

That's literally called an **autoencoder**, and it's been around for decades. The encoder learns a useful compressed representation as a side effect of the reconstruction task. No labels are needed — the image is its own supervision. (This is **unsupervised learning**, which is worth noting because most of what we've done this year has been supervised.)

Autoencoders work fine for what they do. They give you compressed representations that you can use for downstream tasks — classification, denoising, anomaly detection. The compressed vectors of similar images tend to be near each other, which gives you a useful "embedding" of the data. None of this is generative, though. We've built a compression tool, not a generator.

So how do we get from "a thing that compresses and decompresses" to "a thing that generates new images?"

---

§3. Generators

Here's the puzzle. We have a trained autoencoder. Decoder takes a 128-dimensional vector, produces a 64×64 face image. Encoder produces such vectors from real images.

Could we just… pick a random 128-dimensional vector and hand it to the decoder?

Try to predict what happens before reading on.

What happens: garbage. The output is some noisy mess that doesn't look like a face. Why?

Because the decoder was only ever trained on vectors that the encoder produced. The encoder, in the course of training, mapped each real face to some specific point in 128-dimensional space. Maybe all the faces landed in some weird twisted region of that space — a thin manifold, or a few disconnected clusters, or who knows what shape. We don't know what region, because we never asked the encoder to use any *particular* region. We just asked it to use *some* region that lets the decoder reconstruct.

When we pick a random 128-dim vector, we're almost certainly landing somewhere the encoder never went, which means somewhere the decoder was never trained. The decoder confidently produces nonsense for those vectors.

Okay, what if we sampled from the *actual* distribution of encoded vectors? Encode the whole training set, look at where the encodings landed, sample from there? In principle yes — but practically, we'd have to fit a high-dimensional density model to those encodings, which is itself a hard generative modeling problem. We'd have just pushed the problem back one level.

Here's the cleaner idea: **make the encoder produce vectors in some distribution we already know how to sample from.** Pick a target distribution that's easy to sample (say, $N(0, 1)$ — the standard normal), and add a training pressure that pushes the encoder toward producing outputs distributed that way.

If we can pull this off, generation becomes easy: sample $z \sim N(0, 1)$, feed it to the decoder, get an image.

This is the core idea of the VAE: don't just compress, **compress into a sampleable space.** The "variational" part of the name refers to a particular way of measuring how far the encoder's output distribution is from our target.

The pieces we now need:

1. A way to ask the encoder to produce outputs distributed like $N(0, 1)$.
2. A way to measure when it's succeeded.
3. Confidence that this doesn't break reconstruction in the process.

Let's tackle them in order.

---

§4. Encoding to a distribution

How do we ask the encoder to produce outputs distributed like $\mathcal{N}(0, 1)$?

Here's a thought: the encoder is a deterministic function. It takes an image, produces a vector. There's nothing random about it. If we run the encoder on a batch of images, we get some specific set of vectors. We could *measure* how Gaussian-distributed those vectors look across the batch, and add that to the loss.

This is actually a real approach — it shows up in Adversarial Autoencoders and Wasserstein Autoencoders. It works. But it requires choosing some metric on distributions (adversarial training? Wasserstein distance? matching moments?), and that metric has to be estimated from a batch, which is statistically tricky in high dimensions.

The VAE takes a different approach. Instead of having the encoder output a *point* and then measuring the distribution of points across a batch, **have the encoder output a *distribution* directly.**

Here's how. For each input x, the encoder produces two vectors: a mean `mu` and a (log of the) variance `log_var`. These are interpreted as the parameters of a Gaussian distribution:

$$q(z \mid x) = \mathcal{N}(\mu, \sigma^2), \qquad \sigma = \exp(0.5 \cdot \texttt{log\_var})$$

Now we have a per-example distribution. Each x produces not a point but a small Gaussian "blob" in latent space. When we want a latent code for x, we *sample* from this Gaussian:

$$z = \mu + \sigma \cdot \epsilon, \qquad \epsilon \sim \mathcal{N}(0, 1)$$

And we feed *that* sample to the decoder.

This setup might look ad-hoc. Let's pause and notice something subtle.

The encoder doesn't "describe" a distribution. The training procedure imposes one. The `fc_mu` and `fc_log_var` layers in the encoder are just linear projections. There's nothing in those layers that forces their outputs to be the parameters of any particular distribution. We *choose* to interpret them as Gaussian parameters by the way we use them downstream. The sampling step $z = \mu + \sigma \cdot \epsilon$ is what makes the Gaussian assumption real. Once z is generated this way, the model has no choice but to be consistent with the Gaussian story, because the gradients flow through that sampling formula.

The first time this hits, it can feel like a sleight of hand. *Wait, you just said these are means and variances, but they're outputs of a `nn.Linear` — what stops them from being garbage?* The honest answer: nothing in the architecture stops it. The training pressure does.

Why the sampling step? Two reasons that take a while to fully appreciate.

Reason 1: a distribution gives us something to push toward a target. If the encoder outputs a single point per x, we can only measure "distribution-ness" across a whole batch. If the encoder outputs a *distribution* per x, we can measure the gap between that per-example distribution and $\mathcal{N}(0, 1)$ using a single, closed-form formula (the KL divergence between two Gaussians, in §5).

Reason 2: it acts as a regularizer. Because z is sampled each time we run a forward pass, the decoder sees a slightly different z for the same x every time. Effectively, the decoder is being asked: "produce x correctly from *any sample within a small Gaussian neighborhood* of μ." That forces the decoder to be smooth — nearby latent points must decode to similar images, because the decoder has been trained on a fuzzy region around each μ rather than a single point.

This second reason is genuinely surprising and worth sitting with. The decoder is never trained on the deterministic encoding of x. It's trained on noisy samples around the deterministic encoding. The randomness isn't a bug — it's the entire mechanism by which the latent space becomes smooth.

There's also the technical wrinkle of being able to compute gradients through the sampling step. Naively, $z \sim \mathcal{N}(\mu, \sigma^2)$ isn't differentiable — the gradient can't flow back through the sampling operation to μ and σ. The trick $z = \mu + \sigma \cdot \epsilon$ (with ϵ drawn from a *fixed* $\mathcal{N}(0,1)$ that we don't differentiate through) puts μ and σ into the deterministic part of the computation. Now we can take $\partial\text{loss}/\partial\mu$ and $\partial\text{loss}/\partial\sigma$ like for any other parameters. This is called the **reparameterization trick**, and it's why the architecture above can be trained at all.

---

§5. The KL term

So we have an encoder that outputs (`mu`, `log_var`) per example. We sample z from $\mathcal{N}(\mu, \sigma^2)$. The decoder reconstructs from z. So far the loss is just reconstruction error.

Now we need to add a pressure that pushes $q(z \mid x)$ toward $\mathcal{N}(0,1)$. The natural measure of "how far one distribution is from another" is the **Kullback-Leibler divergence**:

$$\mathrm{KL}(q \| p) = \mathbb{E}_{z \sim q}[\log q(z) - \log p(z)]$$

KL is non-negative, equal to zero exactly when the two distributions are identical. We want $\mathrm{KL}(q(z \mid x) \| \mathcal{N}(0,1))$ to be small for every x.

For two Gaussians, KL has a closed form. For $q(z \mid x) = \mathcal{N}(\mu, \sigma^2)$ and $p(z) = \mathcal{N}(0,1)$:

$$\mathrm{KL} = -\frac{1}{2} \sum (1 + \texttt{log\_var} - \mu^2 - \exp(\texttt{log\_var}))$$

(summed over the latent dimensions). This is computable in a single line of code. No sampling, no estimation — closed-form formula in the encoder's outputs.

So our total VAE loss is:

$$L = \text{reconstruction loss} + \beta \cdot \text{KL term}$$

where β is a hyperparameter. $\beta = 1$ corresponds to the "true" theoretical objective (a lower bound on the log-likelihood of the data, called the ELBO; we won't derive it but it's worth knowing it exists). Smaller β puts more weight on reconstruction; larger β puts more weight on matching the input.

That all sounds reasonable. But let's stop and look at it carefully, because something seems off.

We're pushing every single $q(z \mid x)$ to be close to $\mathcal{N}(0,1)$. But if literally every per-example distribution were $\mathcal{N}(0,1)$, then encoding any image would produce the same distribution as encoding any other image. The decoder would receive what amounts to fresh noise regardless of input. *How could reconstruction possibly work?*

Try to resolve this puzzle before reading on.

Here's the resolution. **In equilibrium, $q(z \mid x)$ is *not* equal to $\mathcal{N}(0,1)$ for any x.** The KL term acts as a pressure pushing toward the input, but the reconstruction term acts as an opposing pressure pulling away from it.

If $\mu = 0$ and $\texttt{log\_var} = 0$ ($\sigma = 1$) for every input — the values that make KL exactly zero — then z is essentially noise unrelated to x, and the decoder can't possibly reconstruct anything. Reconstruction loss explodes. So the optimizer can't sit there. It has to find a compromise.

The compromise looks like this: for each x, μ shifts away from zero just enough to encode useful information about x. σ stays *small* (well below 1) so that the blob is localized — nearby samples around μ all roughly correspond to the same x. Each individual $q(z \mid x)$ is a small Gaussian located *somewhere within* the $\mathcal{N}(0, 1)$ region, not equal to it.

The blobs from different x's land at different locations in latent space. That's how the decoder distinguishes them.

This is the picture worth carrying around: Imagine $\mathcal{N}(0, 1)$ as a large bell-shaped region covering some neighborhood of the origin in latent space. Each x's blob is a *small marble* — small in spread, located at a specific position within the bell. Together, the marbles from the whole training set scatter across the bell-shaped region and approximately fill it out. Individually, no marble is the whole bell.

When we sample $z \sim \mathcal{N}(0, 1)$ at generation time, we're picking a random point in the bell. There's almost always *some* marble (or a region covered by overlapping marbles) near that point, and the decoder has been trained to produce a plausible image there.

This picture also explains why the system works, but it's worth asking: does the math *guarantee* the marbles fill the bell? Or might they cluster in one part and leave the rest empty?

---

§6. The aggregate-vs-per-example issue (optional deeper dive)

The KL term in our loss measures $\mathbb{E}_x[\mathrm{KL}(q(z \mid x) \| \mathcal{N}(0, 1))]$ — the *expected per-example* KL. But what we *want* for generation to work is for the **aggregate** distribution

$$q(z) = \mathbb{E}_x[q(z \mid x)]$$

(the mixture you get if you encode the whole dataset) to look like $\mathcal{N}(0, 1)$. These are not the same thing.

There's a nice identity that connects them. It can be shown that:

$$\mathbb{E}_x[\mathrm{KL}(q(z \mid x) \| p(z))] = I(x; z) + \mathrm{KL}(q(z) \| p(z))$$

where $I(x; z)$ is the mutual information between input and latent under the encoder's distribution. The proof is a one-line manipulation: split $\log(q(z \mid x)/p(z))$ into $\log(q(z \mid x)/q(z)) + \log(q(z)/p(z))$, take expectations, and the first piece is mutual information while the second is aggregate KL.

This identity is illuminating because it shows what the KL term is *really* doing:

- It's pushing **mutual information** down (toward zero information about x in z).
- It's pushing **aggregate KL** down (toward the prior).

Both at once.

Reconstruction loss pushes back on mutual information — the decoder needs z to carry information about x to reconstruct. So in equilibrium, mutual information stays positive (above whatever threshold reconstruction requires), and aggregate KL is non-zero (we can't drive the whole KL term to zero without destroying mutual information).

The consequence: the aggregate $q(z)$ is approximately $\mathcal{N}(0, 1)$, but not exactly. There are "holes" in the aggregate — regions where the prior puts mass but few encoded points actually land. This is called the **prior hole problem** or **aggregate posterior gap**, and it's a well-known limitation of VAEs.

You can see the consequences in samples: VAE-generated images sometimes look "off" in subtle ways, and the worst samples come from regions of the prior that are far from any cluster of training encodings. This is

part of why VAE samples have a characteristic blurriness/blandness beyond what reconstruction MSE alone would predict.

A lot of post-VAE research (VampPrior, VQ-VAE, AAE, WAE, diffusion models themselves) is essentially trying to address this gap. The VAE is the simplest member of a family, not the final word.

For our purposes the takeaway is: **the convergence to $\mathcal{N}(0,1)$ is approximate and bounded, not guaranteed and tight.** The KL term in our loss is doing real work — it's bounding the aggregate KL from above — but it leaves a residual gap. The intuition "the loss nudges things toward $\mathcal{N}(0,1)$" is precisely correct; what's bounded is *how far* the aggregate can wander.

---

§7. KL annealing: a practical wrinkle

There's a subtle failure mode worth knowing about. If we start training with $\beta = 1$ from the very first batch, the optimizer can find a local minimum we don't want: every $q(z \mid x)$ collapses exactly to $\mathcal{N}(0,1)$, the decoder learns to produce a generic "average face" regardless of z, and reconstruction is bad but the KL term is zero so the total loss is finite. Once the model is sitting in this collapsed solution, it has trouble escaping.

This is called **posterior collapse**, and it's a real problem with powerful decoders (especially text decoders that can fake plausible outputs without conditioning on z).

The workaround: start with $\beta = 0$ (pure reconstruction loss, the model is just an autoencoder), and gradually ramp β up to 1 over the first several epochs. The encoder is forced to use μ meaningfully because that's the only way to drive reconstruction down. By the time KL pressure kicks in, the encoder is already in a regime where "use μ informatively" is the dominant solution, and posterior collapse is no longer an easy escape.

This is called **KL annealing**, and you'll see it in the training loop.

---

§8. The decoder: getting from a vector back to an image

We've spent a lot of time on the encoder side. The decoder is conceptually simpler but architecturally interesting.

The shape problem: the encoder went from `64 × 64 × 3` (a 12,288-dim image) down to a 128-dim latent vector. The decoder has to go in reverse — from a 128-dim vector back to a $64 \times 64 \times 3$ image. We need to *grow* spatial size.

How do we grow spatial size with a convolutional network?

A regular convolution with stride 2 *shrinks* spatial size (halves it). We need the opposite operation: something that *doubles* spatial size while doing a learned local operation. There's such a thing, and it's called a **transposed convolution** (sometimes also "deconvolution," though that name is misleading and the field has mostly stopped using it).

Mechanically: a transposed convolution with stride 2 takes each input cell and writes its kernel-sized footprint into a 2× larger output, with overlapping regions summing. The result is an upsampling operation with learnable weights — the network learns *how* to upsample, rather than using a fixed rule like nearest-neighbor or bilinear interpolation.

Practical note: transposed convolutions can produce a characteristic **checkerboard artifact** when the stride doesn't divide the kernel size evenly. You may see faint checkerboard patterns in early-training samples. Modern code often uses "upsample then convolve" instead (nearest-neighbor upsample to 2× size,

then a regular stride-1 conv) to avoid this. For our notebook we'll use transposed convolutions because they're simpler to read and the artifact is mild at 64×64.

The decoder architecture mirrors the encoder. The encoder went through four stride-2 convolutions, shrinking spatial size from 64 to 4 ($64 \to 32 \to 16 \to 8 \to 4$). The decoder does the reverse: starts with a linear projection from the 128-dim latent to a $4 \times 4 \times 256$ feature volume, then four stride-2 transposed convolutions grow spatial size back up ($4 \to 8 \to 16 \to 32 \to 64$). The channel counts also reverse.

This mirror-image structure isn't required — you could in principle design any encoder/decoder pair — but it's the cleanest, most symmetric choice and works well.

---

§9. Putting it all together

Architecture:

- **Encoder:** four stride-2 convolutions on the image, then two parallel linear heads producing `mu` and `log_var`.
- **Reparameterization:** $z = \mu + \exp(0.5 \cdot \texttt{log\_var}) \cdot \epsilon$, with $\epsilon \sim \mathcal{N}(0, 1)$.
- **Decoder:** linear projection to spatial feature volume, then four stride-2 transposed convolutions back up to image resolution, with Tanh final activation matching the [-1, 1] input normalization.

Loss:

- **Reconstruction:** MSE between original and reconstructed image, summed over pixels.
- **KL:** closed-form expression in `mu` and `log_var`, summed over latent dimensions.
- **Total:** reconstruction + $\beta \cdot$ KL, with β annealed from 0 to 1 over the first several epochs.

What to expect when you run the notebook:

- The loss drops fast at first while β is small (basically just an autoencoder phase).
- When β ramps up, you'll see a visible inflection in the loss curve — the model has to start balancing the two pressures.
- Reconstructions get sharper over epochs but stay somewhat blurry. This blurriness isn't a bug you can fix by training longer; it's a fundamental consequence of MSE loss and the blob-not-point encoding (the decoder has to handle a neighborhood around each μ, and ends up producing the *average* of plausible images for that neighborhood — averaging plausible images produces blur).
- Generated samples (sampling $z \sim \mathcal{N}(0, 1)$ and decoding) will look like faces, but with the characteristic VAE-blurriness. They'll be visibly less sharp than the reconstructions.
- Interpolations between two faces will produce smooth morphs through plausible intermediate faces. This is the payoff of the structured latent space — interpolation in latent space corresponds to plausible-image interpolation in pixel space.
- Varying a single latent dimension will sometimes reveal interpretable axes: hair color, lighting direction, pose, age. The model wasn't told about these attributes; they emerge from the structure of the data.

A few questions worth thinking about as you work through the notebook:

What happens if you set $\beta = 0$ for the entire run? The model becomes a regular autoencoder. Reconstructions should get sharper. Generated samples ($z \sim \mathcal{N}(0, 1)$ and decode) should become much worse — possibly unrecognizable noise. Why? Because the encoder is free to use any region of latent space, and $\mathcal{N}(0, 1)$ is unlikely to land where it actually put things.

What would happen with $\beta = 10$ instead of $\beta = 1$? More compression toward the prior. Reconstructions would get blurrier, but generated samples might look *more diverse* because the aggregate is closer to a full $\mathcal{N}(0, 1)$. This is the trade-off space of "β-VAEs."

The fundamental tension: sharp reconstruction wants to use the latent space freely; sampleable generation wants the latent space compressed toward $\mathcal{N}(0, 1)$. The β balances these. There's no setting where both win without limit — the trade-off is real.

---

§10. Looking ahead

The VAE works. It generates faces. The faces are blurry, the latent space isn't quite $\mathcal{N}(0, 1)$, and there are theoretical loose ends about the aggregate vs. per-example distinction. None of these are accidents — they're consequences of the choices the VAE makes, and each is a starting point for a research direction:

- **Sharper samples:** the blurriness comes from MSE-loss + averaging-over-blob. Diffusion models fix both by replacing the single-shot encode/decode with many small denoising steps. We'll see this next.
- **Better latent fit:** VampPrior, VQ-VAE, and similar models replace the fixed $\mathcal{N}(0, 1)$ prior with a learned one that better matches what the encoder actually produces.
- **Direct aggregate matching:** Adversarial Autoencoders and Wasserstein Autoencoders skip the per-example KL and directly minimize a distance between the aggregate posterior and the prior. Sharper samples, harder training.

For now, sit with the VAE. The two ideas — *encoding to a distribution* and *KL pressure toward a sampleable prior* — are the conceptual core of most modern generative modeling, including diffusion. If you understand the VAE in depth, the next several units will feel like elaborations on themes you already know.

When you read through the notebook, try to keep the mental picture:

- A scatter of small Gaussian "blobs" tiling a region of latent space.
- Each blob the encoding of one training image.
- The decoder trained to produce that image from any sample inside the blob.
- The blobs collectively shaped to approximately fill $\mathcal{N}(0, 1)$.
- Generation = drop a point into the bell-shaped region and decode.

If you can hold that picture and explain why each piece of the loss is there, you've got the VAE.