

CS 1810 Spring 2026 Section 8

Deep Learning Medley: Self-Supervised Learning and Generative Modeling, Contrastive Learning, GANs, Energy-Based Models

1 Motivation and Outline

Until now, the models that we have discussed mainly focus on solving tasks like regression and classification using discriminative models, which model the relationship between a target output y and some input features x . These lecture notes begin to dive into new paradigms of deep learning, throughout which we will commonly be aiming to learn a latent representation z of an input x , though what objective we use to learn this representation and how we will use the representations will vary across model architectures.

2 Generative modeling via VAEs

2.1 Reviewing generative modeling and vanilla autoencoders

Reviewing the meaning of generative modeling. While a discriminative model learns the boundary between classes (i.e., $p(y \mid x)$), a generative model learns the underlying distribution of the data itself (i.e., $p(x)$). So, e.g., a well-calibrated generative model allows us to:

1. **Generate new datapoints:** By sampling from $p(x)$, we can create novel, synthetic data that is indistinguishable from the training data.
2. **Detect out-of-distribution inputs:** If a new datapoint x_{new} yields a very low $p(x_{new})$, we can flag it as anomalous.
3. **Extract compact latent representations:** Generative models often funnel data through a bottleneck, yielding a low-dimensional vector z that captures the semantic essence of the input.

The standard autoencoder. Before diving into VAEs, we revisit the standard autoencoder, which is a staple of self-supervised learning. In autoencoding, we learn an encoder f_ϕ and a decoder g_θ to compress an input x into a latent z .

- **Forward pass:** The input x is passed through the encoder to produce the latent vector $z = f_\phi(x)$. The decoder then attempts to map z back to the original input space to produce $\hat{x} = g_\theta(z)$.
- **Loss function:** We optimize the mean squared error (MSE) between the input and the output:

$$\mathcal{L}_{\text{MSE}} = \|\hat{x} - x\|^2.$$

So, autoencoders are useful for obtaining compact latent representations. We have already seen dimensionality reduction via PCA, and, for example, a fully linear autoencoder would recover PCA. But of course, autoencoders can be much more general and expressive in learning latent representations.

Why the vanilla autoencoder is not a successful generative model. While standard autoencoders excel at dimensionality reduction, they fail as generative models. We might try to generate a new image by sampling a random point z in the latent space, e.g., maybe we could guess $z \sim \mathcal{N}(0, I)$, and then passing it through the decoder. But for vanilla autoencoders, the result is usually unrecognizable noise.

This happens because the latent space of a standard autoencoder is unconstrained. Without knowing the structure or “range” of the latent space, the network does not enforce any continuous structure on z ; it simply places encoded points wherever it minimizes the MSE, often leaving vast “empty” regions in the latent space. So if we want to take random latent vectors z and decode them to generate samples, then the challenge is that we don’t know how to choose z .

2.2 Overviewing VAEs

The above discussion leads into how we construct variational autoencoders (VAEs).

Overall structure of VAEs: A probabilistic perspective on the encoder and decoder. To construct a true generative model $p(x)$ using the autoencoder architecture, we must shift to a probabilistic perspective. Before we viewed the encoder and decoder as just deterministic functions on the input and latent space, respectively. Now, we pretend that the data is generated from a latent variable model with latent z .

Specifically, we assume that the data is generated as

$$p_{\theta}(x, z) = p(z)p_{\theta}(x | z),$$

for some latent z which has prior $z \sim \mathcal{N}(0, 1)$; it is hard to say exactly what z corresponds to intuitively, but what we are assuming is that there is some way to encode the underlying variation in our data into a latent variable which has this standard Normal structure. The reason that we choose standard Normal is that it is simple, so that later we can easily sample arbitrary points z in the latent space, and use the decoder network to generate samples from the original data distribution.

Now after assuming this data structure, our goal becomes to learn

- an **encoder**, which given an input x aims to approximate the probability distribution $p(z | x)$ over the latent space, given x . We do not directly make assumptions on the shape of this in our probabilistic model for the data, but as we will soon discuss, what we do is that we let our encoder learn a distribution $q(z | x)$ which can be thought of as a Gaussian approximation to $p(z | x)$. So, really what the encoder outputs is the parameters of a Gaussian distribution, i.e., a mean vector μ and variance vector σ^2 whose dimensionalities are both the dimension of the latent space.
- a **decoder**, which takes as input a latent z , and aims to approximate the distribution $p(x | z)$ over the input space, given z . We also assume in our model that this $p(x | z)$ is Gaussian, and we often also assume that the variance is a fixed constant σ^2 , so what the decoder really outputs is $\mu(z)$ in the input space, which is the mean of $p(x | z)$.

This $\mu(z)$ corresponds to reconstructing the original x , which was passed through the encoder to obtain the latent z in the first place. Of course, we can also randomly sample any latent point from the $\mathcal{N}(0, I)$ distribution, and decode it to obtain something in the input space.

2.3 VAE training objective

We have now overviewed the structure of VAEs and what we want each of the encoder and decoder to do. It is still unclear, however, how to train the VAE.

The log-likelihood objective and intractability. To train VAEs, the natural objective is to maximize the likelihood of our training data. That is, we want to maximize the log-likelihood

$$\max \mathbb{E}_{x \sim p_d} [\log p(x)],$$

where p_d is the data distribution (over the input space), and $p(x)$ is the model distribution defined by the VAE. But it is uncertain how to compute $p(x)$. We can express the marginal likelihood $p(x)$ by the law of total probability, integrating over all possible latent variables z , obtaining

$$p(x) = \int p(x | z) p(z) dz.$$

But this leads to an intractability problem: optimizing this integral directly is impossible, because it is computationally intractable to compute the integral over all latent z in a high-dimensional space. (We also cannot simply sample a few random z 's to estimate it, because for almost all random z 's, the likelihood of generating our specific image x , $p(x | z)$, is practically zero.)

Variational Inference and the ELBO. Because exact inference is intractable, we turn to variational inference. The core idea, as we previewed above, is to use our encoder to learn a new distribution $q(z | x)$ which is called the variational distribution, and which tries to approximate the true $p(z | x)$ by a Gaussian distribution. This is also important because otherwise, without making parametric assumptions, it is unclear how our encoder will be “learning” the distribution $p(z | x)$; now, the encoder just needs to learn the parameters of the Gaussian distribution. (As a technical detail, this is also making the assumption that the latent space indices are conditionally independent given x .)

So, the key idea is that we will use the $q(z | x)$ to derive a lower bound on the log-likelihood, which we can then train our model to optimize.

Deriving the evidence lower bound. We will derive the variational bound of likelihood step-by-step. Let's start with the log-likelihood of a single datapoint x :

$$\log p(x) = \log \int p(x|z)p(z)dz.$$

We multiply the inside of the integral by $1 = \frac{q(z|x)}{q(z|x)}$:

$$\log p(x) = \log \int q(z|x) \frac{p(x|z)p(z)}{q(z|x)} dz.$$

We can rewrite this integral as an expectation over our variational distribution $q(z|x)$:

$$\log p(x) = \log \left(\mathbb{E}_{z \sim q(z|x)} \left[\frac{p(x|z)p(z)}{q(z|x)} \right] \right).$$

Because the logarithm is a strictly concave function, we can apply **Jensen's Inequality** ($\log(\mathbb{E}[Y]) \geq \mathbb{E}[\log(Y)]$) to push the log inside the expectation:

$$\log p(x) \geq \mathbb{E}_{z \sim q(z|x)} \left[\log \left(\frac{p(x|z)p(z)}{q(z|x)} \right) \right]$$

Using the properties of logarithms, we separate the terms:

$$\log p(x) \geq \mathbb{E}_{z \sim q(z|x)} [\log p(x|z)] + \mathbb{E}_{z \sim q(z|x)} \left[\log \left(\frac{p(z)}{q(z|x)} \right) \right].$$

The second term is exactly the negative **Kullback-Leibler (KL) Divergence** between the approximate posterior $q(z|x)$ and the prior $p(z)$. Thus, we arrive at the final expression:

$$\log p(x) \geq \mathbb{E}_{z \sim q(z|x)} [\log p(x|z)] - KL(q(z|x) || p(z)).$$

Understanding the ELBO components. This final expression is known as the *Evidence Lower Bound (ELBO)*. It essentially states that we can optimize a lower bound on the data likelihood by drawing latents from a variational distribution $q(z|x)$ and maximizing the conditional likelihood $p(x|z)$ on those specific latents.

Let's break down the two opposing forces in this objective:

1. **Reconstruction Term:** $\mathbb{E}_{z \sim q(z|x)} [\log p(x|z)]$

This term encourages the decoder to accurately reconstruct the input x from the sampled latent z . In practice, if we assume $p(x|z)$ is a Gaussian with a fixed variance, maximizing this term is mathematically equivalent to minimizing the Mean Squared Error (MSE).

2. **KL-Divergence Term:** $KL(q(z|x) || p(z))$

This acts as an information bottleneck and regularizer. It penalizes the encoder if its predicted distribution $q(z|x)$ drifts too far from our prior $p(z) = \mathcal{N}(0, 1)$. It forces the latent space to pack tightly around the origin and maintain a Gaussian shape.

When $q(z|x)$ exactly matches the true posterior distribution $p(z|x)$, the KL divergence between them is zero, the evidence lower bound is perfectly tight, and the right and left sides on the equation above are equal.

Putting it together into the VAE. So, a VAE is roughly an autoencoder architecture trained to maximize the ELBO. To reiterate,

- We parameterize the variational distribution $q(z|x)$ using the **encoder**. Instead of outputting a single vector z , the encoder outputs the parameters of a Gaussian distribution: a mean vector μ and a variance vector σ^2 .
- We set the prior distribution $p(z)$ to be the standard normal, $\mathcal{N}(0, 1)$.
- We parameterize the conditional distribution $p(x|z)$ using the **decoder**, which maps a sampled z back to the image space.

The VAE training loss is decomposed into the negative of the ELBO (since we want to minimize loss in PyTorch):

$$\mathcal{L}_{VAE} = \mathcal{L}_{\text{recon}}(\hat{x}, x) + \mathcal{L}_{KL}(q(z|x) || p(z))$$

Latent space packing and interpolation. The primary visual difference between a standard Autoencoder and a VAE lies in the geometry of the latent space.

In a VAE, notice how the KL-Divergence packs everything nicely inside the Normal Distribution. Because the latents are tightly packed around the origin in a continuous standard normal distribution $\mathcal{N}(0, I)$, we achieve our generative goal: sampling from $\mathcal{N}(0, 1)$ yields recognizable digits!

Furthermore, this packed, continuous structure allows us to create a **generative map**. We can smoothly interpolate across the latent space, e.g., for MNIST, we can take a straight line between the latent vector for a “3” and a “7”. As we move along this line and decode the points, we can hope to see a smooth transition from one digit to the other. (However, whether the transition is semantically meaningful is not so certain.)

2.4 Training VAEs: The reparameterization trick

We now encounter a critical implementation hurdle. To train the VAE via gradient descent, we need to backpropagate the loss through the entire network, including the sampling step $z \sim q(z|x)$.

The problem: How do we differentiate through a random sample? If we simply sample z using a standard random number generator (e.g., `torch.normal(mu, sigma)`), that operation blocks the flow of gradients. A random sampling node has no mathematical derivative with respect to the encoder’s parameters (μ and σ).

The deterministic solution. We differentiate through samples from the variational distribution $q(z|x)$ by using the **reparameterization trick**. We express the random sample as a deterministic transformation of an auxiliary, parameter-free noise variable.

Mathematically, instead of sampling $z \sim \mathcal{N}(\mu, \sigma^2)$ directly, we first sample an independent noise variable from the standard normal prior:

$$\epsilon \sim \mathcal{N}(0, I)$$

And then we deterministically scale and shift this noise to compute z :

$$z = \mu + \sigma \odot \epsilon$$

(where $\odot$ represents element-wise multiplication).

Now, the stochasticity is entirely isolated in ϵ , which has no parameters that require updating. The variables μ and σ (the outputs of our encoder neural network) are now part of a differentiable, deterministic equation. PyTorch’s autograd engine can now easily compute the gradients:

$$\begin{aligned}\frac{\partial z}{\partial \mu} &= 1 \\ \frac{\partial z}{\partial \sigma} &= \epsilon\end{aligned}$$

Computing the KL Divergence analytically. Because we chose both our prior $p(z)$ and our posterior approximation $q(z|x)$ to be Gaussian, we do not need to approximate the KL-Divergence

term via sampling. It has a closed-form analytical solution. For a D -dimensional multivariate Gaussian with diagonal covariance, the KL divergence to $\mathcal{N}(0, I)$ is:

$$\mathcal{L}_{KL} = -\frac{1}{2} \sum_{j=1}^D (1 + \log(\sigma_j^2) - \mu_j^2 - \sigma_j^2)$$

This formula is highly efficient to compute during the forward pass.

Exercise: Analyzing VAE Bugs

Imagine you are implementing a Variational Autoencoder (VAE) in code to generate images of handwritten digits. Your model consists of an Encoder (which outputs the distribution parameters μ and $\log(\sigma^2)$) and a Decoder (which takes a latent vector z and outputs an image). However, you make a few coding mistakes. Analyze the following three buggy implementations. For each bug, explain exactly what will go wrong during training and inference.

- Implementation A (The Sampling Bug):** You forget the reparameterization trick. To get z , you simply write a standard coding command:
`z = torch.normal(mean=mu, std=torch.exp(0.5 * logvar)).`
- Implementation B (The Deterministic Bug):** You decide to skip sampling entirely to "simplify" things. You simply set $z = \mu$ and pass that directly to the decoder. You still calculate both the MSE and KL-Divergence loss.
- Implementation C (The Prior Bug):** You use the correct reparameterization trick, but you accidentally delete the KL-Divergence term from your loss function, leaving only the Reconstruction (MSE) loss.

Solution:

- Implementation A (The Sampling Bug):**
Backpropagation will fundamentally fail for the Encoder. The computational graph is broken at the stochastic `torch.normal` step. While the Decoder will receive gradients and attempt to learn a mapping from noise to images, the Encoder's weights will never be updated by the optimizer. The model will not learn a meaningful latent representation.
- Implementation B (The Deterministic Bug):**
If $z = \mu$, the model never experiences any variance or noise during training. The decoder becomes overly reliant on exact, infinitesimally precise points in the latent space. Even though the KL-Divergence loss is pushing μ toward 0, the lack of variance (σ) means the decoder won't learn to decode a *region* of space smoothly. If you try to sample $z \sim \mathcal{N}(0, 1)$ at test time, the model will likely generate poor, artifact-heavy images because it was never trained to handle variance around μ . It essentially becomes a regularized Autoencoder rather than a robust probabilistic Generative Model.
- Implementation C (The Prior Bug):**
Without the KL-Divergence term acting as a regularizer, the VAE degrades back into a standard autoencoder (specifically, an autoencoder with injected noise). The latent space will

not be forced to pack into the Standard Normal range $\mathcal{N}(0, I)$. The latents will spread out arbitrarily far to minimize the reconstruction error as easily as possible. At inference time, it becomes impossible to sample new, valid images because we do not know the shape or limits of the unconstrained latent distribution.

3 From Denoising Autoencoders to Diffusion Models

We've discussed how variational inference converts an autoencoder into a generative model. Now we look at a second pathway: converting self-supervised algorithms to generative models through **iterative sampling**.

3.1 The Denoising Autoencoder

In denoising autoencoders, we modify an autoencoder by corrupting the input x (e.g., by masking certain pixels or by adding Gaussian noise) and training the network to reconstruct the clean input.

Geometrically, a denoising autoencoder learns the local "shape" of the data manifold. If a data point is pushed off the manifold by noise, the autoencoder learns a vector field $g_{\theta'}(f_{\theta}(\tilde{x}))$ that points the corrupted sample $q_{\mathcal{D}}(\tilde{x}|x)$ back toward the high-density manifold of true data.

3.2 Diffusion Models (Continuous Denoising)

How can we generate samples using a denoising autoencoder? We can initialize a sample from pure noise and repeatedly use the denoising autoencoder to denoise the sample step-by-step. When the noise added is Gaussian noise, this reduces exactly to a **Diffusion Model**.

In diffusion models, we define a forward process that takes a datapoint x and forms a corrupted sample at timestep t :

$$\tilde{x}_t = \sqrt{\bar{\alpha}_t}x + \sqrt{1 - \bar{\alpha}_t}\epsilon$$

where $\epsilon \sim \mathcal{N}(0, I)$ is sampled Gaussian noise, and $\bar{\alpha}_t$ represents a noise schedule.

- **Training:** The network (typically a U-Net acting as our denoising autoencoder) is trained to reconstruct the original sample x (or predict the added noise ϵ) with a simple training loss: $\mathcal{L}_{MSE} = \|\hat{x} - x\|^2$.
- **Generation (Reverse Process):** To generate from the denoising autoencoder, we start with a sample x_T drawn from pure Gaussian noise. At each iteration of generation, we take the previous prediction, and use the network to remove a small amount of noise, effectively stepping backward in time from t to $t - 1$.

This iterative denoising process in Diffusion Models allows the model to gradually transform static noise into incredibly high-fidelity images. Because the task is broken down into hundreds of tiny, simple denoising steps, Diffusion Models currently achieve State-of-the-Art (SOTA) generation quality, surpassing VAEs in image sharpness.

4 Masked Language Modeling (Discrete Diffusion)

We can apply the exact same iterative sampling philosophy to discrete text data, leading to the architectures that power modern NLP.

In **Masked Language Modeling (MLM)**, we take a datapoint x (a sentence) and form a corrupted sample $\tilde{x}$ by masking out portions of tokens of the sentence. We learn a denoising autoencoder that predicts the values of the masked tokens.

4.1 BERT Architecture Details

For models like BERT (Bidirectional Encoder Representations from Transformers), the standard procedure is to randomly mask 15% of tokens in a sequence.

For example, the input sequence:

"Let's stick to improvisation in this skit"

becomes corrupted to:

"Let's stick to [MASK] in this skit"

We pass this corrupted sequence through the network and use the output representation at the masked word's position to predict the original word. This is done using a Feed-Forward Neural Network followed by a Softmax layer ($FFNN + Softmax$). The possible classes for the prediction are all the words in the English vocabulary.

4.2 Generation via Discrete Iteration

While BERT is mostly used as a self-supervised pre-training step to learn representations, we can generate from the denoising model. By initializing a sequence from scratch (e.g., all [MASK] tokens) and using the denoising autoencoder to unmask tokens one at a time, we iteratively build a sentence. This process corresponds to "discrete diffusion," utilized in models like MaskGIT.

Concept Check: Modality Differences

Compare the corruption process used in image Diffusion Models versus the one used in Masked Language Models (like BERT). Why do we use different corruption strategies for these different modalities?

Solution:

- **Diffusion (Images):** Uses continuous Gaussian noise addition. Pixel values in an image are continuous numerical variables (e.g., RGB values normalized from -1.0 to 1.0). Adding small amounts of Gaussian noise smoothly moves the data point off the manifold.
- **MLM (Text):** Uses discrete masking (substituting a token with a special [MASK] token). Text is represented by discrete tokens (words or subwords) mapped to integer IDs from a fixed vocabulary. You cannot add "continuous noise" to a discrete integer ID (e.g., the token ID for "apple" + 0.5 noise does not yield a mathematically valid word). Therefore, the discrete equivalent of "corrupting" the data is to hide the categorical variables entirely.

5 Contrastive Learning Revisited: A Classification Perspective

We now revisit contrastive learning, but with a more precise lens: *contrastive learning can be interpreted as a classification problem*. This perspective will be crucial for connecting contrastive methods to GANs and energy-based models.

Contrastive Representation Learning Setup Recall that in contrastive learning, our goal is to learn a representation $z = f_\phi(x)$ such that:

- Similar (positive) pairs are close in representation space.
- Dissimilar (negative) pairs are far apart.

To construct this learning signal, we define:

- A positive pair: (x, x') , where x' is an augmented version of x .
- A set of negative samples: $\{x_k\}$ drawn from other datapoints.

So, given an anchor x_i , we encode all samples:

$$z_i = f_\phi(x_i), \quad z_{\text{pos}} = f_\phi(x'), \quad z_k = f_\phi(x_k),$$

and then we define a similarity function, typically cosine similarity, which is

$$\text{sim}(z_i, z_j) = \frac{z_i^\top z_j}{\|z_i\| \|z_j\|}.$$

InfoNCE Loss as a Classification Objective. The standard contrastive loss is the InfoNCE loss:

$$\mathcal{L} = -\log \left(\frac{\exp(\text{sim}(z_i, z_{\text{pos}})/\tau)}{\sum_{j \in \{\text{pos} \cup \text{negatives}\}} \exp(\text{sim}(z_i, z_j)/\tau)} \right),$$

where $\tau > 0$ is a temperature parameter. The key insight is that this is a softmax classification problem. We can interpret this as a $(K + 1)$ -class classification problem (1 positive + K negatives), where the model must identify which candidate is the true match for x_i .

Thus, contrastive learning is equivalent to learning a classifier that selects the correct positive among a set of candidates.

The Role of Negative Samples. The effectiveness of contrastive learning depends critically on the *negative distribution*.

If negatives are trivial (e.g., Gaussian noise), the model can solve the task using superficial cues, e.g., “structured image $\rightarrow$ positive”, and “noise $\rightarrow$ negative”. And naturally, this does not really help refine our representations. To learn meaningful features, negatives must be close to the data manifold, and semantically similar but incorrect. This forces the model to learn higher-level structure (shape, texture, semantics).

In high-dimensional spaces, naturally occurring hard negatives are rare. This motivates the idea: *Can we **learn** a distribution of hard negatives?* This question leads directly to GANs.

6 From Contrastive Learning to GANs

Learning the Negative Distribution. Instead of sampling negatives from the dataset, we introduce a second neural network to generate negatives. So, we define two networks:

- Discriminator $D_\psi(x)$: predicts whether x is real
- Generator $G_\theta(z)$: maps noise $z \sim p(z)$ to fake samples

Now the contrastive setup becomes:

- Positive samples: $x \sim p_{\text{data}}$
- Negative samples: $x \sim p_G$

The generator plays the role of a learned negative sampler, producing increasingly realistic (hard) negatives, so then the discriminator must learn increasingly refined distinctions. This creates an adversarial game, i.e., where the discriminator improves classification, and the generator improves sample realism.

GAN objective functions. The discriminator is trained as a binary classifier,

$$\mathcal{L}_D = \mathbb{E}_{x \sim p_{\text{data}}} [-\log D(x)] + \mathbb{E}_{z \sim p(z)} [-\log(1 - D(G(z)))].$$

Meanwhile the generator aims to fool the discriminator,

$$\mathcal{L}_G = \mathbb{E}_{z \sim p(z)} [-\log D(G(z))].$$

To interpret this, note that the generator is explicitly optimizing for hard negatives: if $D(G(z))$ is large, the sample looks realistic. Thus, the generator pushes mass toward the data distribution

GANs as Contrastive Learning. We can reinterpret GANs as contrastive learning, with the positive distribution being $p_{\text{data}}(x)$, and the negative distribution being $p_G(x)$. Then, the discriminator acts as a contrastive classifier, while the generator learns the negative distribution.

Practical considerations. Some strengths of GANs are that they produce sharp, high-fidelity samples. Weaknesses include training instability (minimax optimization), mode collapse (limited diversity from the generator), vanishing gradients.

7 From GANs to Energy-Based Models

We now move to a unifying probabilistic perspective.

7.1 Preliminaries and defining the energy function

The Optimal Discriminator and Density Ratios. Examining the GANs discriminator loss, note that for a fixed generator, the optimal discriminator satisfies

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

Taking logits,

$$\text{logit}(D^*(x)) = \log p_{\text{data}}(x) - \log p_G(x).$$

Essentially, the discriminator is roughly capturing the unnormalized log probability of the data, known as the “energy function.”

From Density Ratios to Energy Functions. Define

$$l_\theta(x) := \text{logit}(D_\theta(x)).$$

If we fix $p_G(x)$ (e.g., Gaussian noise), then:

$$\log p_{\text{data}}(x) = l_\theta(x) + \log p_G(x).$$

Thus, we can recover the data density up to normalization. For notation, next define the energy function as

$$E_\theta(x) = -\log p_{\text{data}}(x).$$

7.2 Energy-Based Models

Given an energy function, an energy-based model (EBM) defines the overall distribution as

$$p_\theta(x) = \frac{1}{Z_\theta} \exp(-E_\theta(x)),$$

where the partition function is

$$Z_\theta = \int \exp(-E_\theta(x)) dx.$$

The intuition is low energy corresponds to high probability, and high energy to low probability. As we saw, the logit of the GAN discriminator is an example of modeling and training the energy function; in general, modeling the energy function $E_\theta(x)$ can be easier than modeling the probability distribution $p_\theta(x)$.

The Partition Function Problem. The main challenge is

$$Z_\theta = \int \exp(-E_\theta(x)) dx,$$

which is intractable in high dimensions. Thus, exact likelihood computation is difficult, and we need some alternative methods. Similar to VAEs, where we can use the ELBO to avoid the need to computing the explicit integral, we can use Monte Carlo sampling to approximate training of the partition function of an EBM.

Training via Log-Likelihood Gradients. It turns out that the gradient of the log-likelihood is

$$\nabla_\theta \log p_\theta(x^+) = -\nabla_\theta E_\theta(x^+) + \mathbb{E}_{x \sim p_\theta} [\nabla_\theta E_\theta(x)].$$

We can interpret this as having two phases:

- *Positive phase:* decrease energy of real data.
- *Negative phase:* increase energy of model samples.

So there is the same push-pull dynamic that we saw with training SimCLR and GANs. Once again, we note that GANs are a special case of EBM training where the negative samples come from a learned generator (instead of MCMC).

Sampling Negative Examples. To perform the negative phase, we must sample from $p_\theta(x)$. We will not dive deep into these, but common strategies include

- Noise Contrastive Estimation
 - Use a fixed noise distribution and train a classifier to distinguish data vs noise
 - Very similar to early contrastive learning formulations
- Generator-Based Negatives (GAN-style)
 - Train a generator $G(z)$ to approximate sampling from low energy regions
 - Very similar to GAN training
- MCMC / Langevin Dynamics
 - Start from noise iteratively follow $-\nabla E(c)$ to move towards low energy regions

EBMs as a unifying view of generative modeling. Using the language we used for EBMs, we can view the modeling techniques we have discussed as follows.

- Contrastive Learning
 - Energy: similarity function
 - Negatives: dataset samples
 - Objective: classify positives vs negatives
- GANs
 - Energy: discriminator logit
 - Negatives: generator samples
 - Objective: adversarial training
- Energy-Based Models
 - Energy: explicitly modeled $E_\theta(x)$
 - Negatives: noise / generator / MCMC
 - Objective: maximize likelihood via push-pull dynamics

The key takeaway is that all three methods share the same core principle of learning an energy landscape where real data has low energy and non-data has high energy.