14
Introduction to Unsupervised learning and Distribution Learning

Much of the book so far concerned supervised learning —i.e., where training dataset consists of datapoints and a label indicating which class they belong to, and the model has to learn to produce the right label given an input. This chapter is an introduction to unsupervised learning, where one has randomly sampled datapoints but no labels or classes. We survey possible goals for this form of learning, and then focus on *distribution learning*, which addresses many of these goals. We also introduce a general idea of energy-based models, which underlies how many researchers approach the modeling problem.

14.1 Possible goals of unsupervised learning

Learn hidden/latent structure of data. An example would be *Principal Component Analysis (PCA)*, concerned with finding the most important directions in the data. Other examples of structure learning can include sparse coding (aka dictionary learning) or nonnegative matrix factorization (NMF).

Learn the distribution of the data. A classic example is Pearson's 1893 contribution to theory of evolution by studying data about the crab population on Malta island. Biologists had sampled a thousand crabs in the wild, and measured 23 attributes (e.g., length, weight, etc.) for each. The presumption was that these datapoints should exhibit Gaussian distribution, but Pearson could not find a good fit to a Gaussian. He was able to show however that the distribution was actually *mixture* of two Gaussians. Thus the population consisted of two distinct species, which had diverged not too long ago in evolutionary terms.

In general, density estimation starts from the hypothesis that

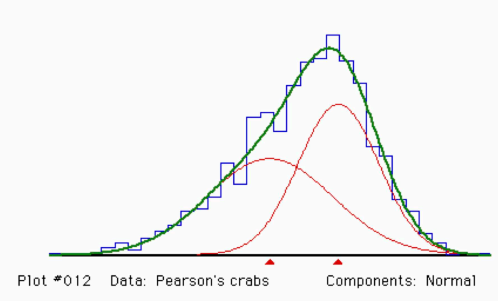

Figure 14.1: Visualization of Pearson's Crab Data as mixture of two Gaussians. (Credit: MIX homepage at McMaster University.)

the unlabeled dataset consists of i.i.d. samples from a fixed distribution, and model θ learns representation of some distribution $p_\theta(\cdot)$ that assigns a probability $p_\theta(x)$ to datapoint x. This is the general problem of *density estimation.*

One form of density estimation is to learn a *generative model*, where the learnt distribution has the form $p_\theta(h, x)$ where x is the observable (i.e., datapoint) and h consists of a vector of hidden variables, often called *latent* variables. Then the density distribution of x is $\int p_\theta(h, x)dh$. In the crab example, the distribution a mixture of Gaussians $\mathcal{N}(\mu_1, \Sigma_1), \mathcal{N}(\mu_2, \Sigma_2)$ where the first contributes ρ_1 fraction of samples and the other contributes $1 - \rho_1$ fraction. Then θ vector consists of parameters of the two Gaussians as well as ρ_1. The visible part x consists of attribute vector for a crab. Hidden vector h consists of a bit, indicating which of the two Gaussians this x was generated from, as well as the value of the gaussian random variable that generated x.

Learning good representation/featurization of data For example, the pixel representation of images may not be very useful in other tasks and one may desire a more "high level" representation that allows downstream tasks to be solved in a data-efficient way. One would hope to learn such featurization using unlabeled data.

In some settings, featurization is learnt via generative models: one assumes a data distribution $p_\theta(h, x)$ as above and the featurization of the visible samplepoint x is assumed to be the hidden variable h that was used to generate it. More precisely, the hidden variable is a sample from the conditional distribution $p(h|x)$. This view of representation learning is used in the *autoencoders* described later.

For example, topic models are a simple probabilistic model of text generation, where x is some piece of text, and h is the proportion of specific topics ("sports," "politics" etc.). Then one could imagine that h is some short and more high-level

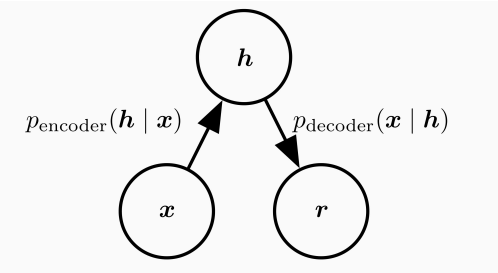

Figure 14.2: Autoencoder defined using a density distribution $p(h, x)$, where h is the latent feature vector corresponding to visible vector x. The process of computing h given x is called "encoding" and the reverse is called "decoding." In general applying the encoder on x followed by the decoder would not give x again, since the composed transformation is a sample from a distribution.

descriptor of x.

Many techniques for density estimation —such as variational methods, described later —also give a notion of a representation: the method for learning the distribution often also come with a candidate distribution for $p(h|x)$. This why students sometimes conflate representation learning with density estimation. But many of today's approaches to representation learning do not boil down to distribution learning.

14.2 *Training Objective for Learning Distributions: Log Likelihood*

We wish to infer the best θ given the set S of i.i.d. samples ("evidence") from the distribution. One standard way to quantify "best" is pick θ is according to the *maximum likelihood principle*, which says that the best model is one that assigns the highest probability to the training dataset.[1]

[1] The maximum likelihood principle is a philosophical stance, not a consequence of some mathematical analysis.

$$\max_{\theta} \prod_{x^{(i)} \in S} p_\theta(x^{(i)}) \tag{14.1}$$

Because log is monotone, this is also equivalent to minimizing the *log likelihood*, which is a sum over training samples and thus similar in form to the training objectives seen so far in the book:

$$\max_{\theta} \sum_{x^{(i)} \in S} \log p_\theta(x^{(i)}) \quad \textit{(log likelihood)} \tag{14.2}$$

Often one uses average log likelihood per datapoint, which means dividing (14.2) by $|S|$.

As in supervised learning, one has to keep track of training log-likelihood in addition to generalization, and choose among models that maximize it. In general such an optimization is computationally intractable for even fairly simple settings, and variants of gradient descent are used in practice.

Of course, the more important question is how well does the trained model learn the data distribution. Clearly, we need a notion of "goodness" for unsupervised learning that is analogous to *generalization* in supervised learning.

14.2.1 Notion of goodness for distribution learning

The most obvious notion of generalization follows from the log likelihood objective. The notion of generalization most analogous to the one in supervised learning is to evaluate the log likelihood objective on *held-out* data: reserve some of the data for testing and compare the average log likelihood of the model on training data with that on test data.

Example 14.2.1. *[Importance of the learner model] The log likelihood objective makes sense for fitting any parametric model to the training data. For example, it is always possible to fit a simple Gaussian distribution $\mathcal{N}(\mu, \sigma^2 I)$ to the training data in $\Re^d$. The log-likelihood objective is*

$$\sum_i \frac{|x_i - \mu|^2}{\sigma^2},$$

which is minimized by setting μ to $\frac{1}{m}\sum_i x_i$ and σ^2 to $\sum_i \frac{1}{n}|x_i - \mu|^2$.

Suppose we carry out this training for the distribution of real-life images. What will we learn? The mean μ will be the vector of average pixel values, and σ^2 will correspond to the average variance per pixel. Thus a random sample from the learn distribution will look like some noisy version of the average of the training images.

This example also shows that matching average log-likelihood for training and held-out data is insufficient for actually learning the distribution. The gaussian model only has $d + 1$ parameters and simple ϵ-cover arguments as in Chapter 5 show under fairly general conditions (such as coordinates of x_i's being bounded) that if the number of training samples is moderately high then the log-likelihood on the average test sample is similar to that on the average training sample. [2] However, the learned distribution may be nothing like the true distribution. To actually begin to learn this complicated distribution we need a more complex model family.

[2] This is reminiscent of the situation in supervised learning whereby a nonsensical model —e.g., one that outputs random labels—has excellent generalization as well because it incurs similar per-point loss on training as well as test data.

But how can we know that log likelihood objective is in principle capable of learning the distribution? The following theorem shows so assuming that the function class used for learning –i.e. the one Q is drawn from–is expressive enough.

Theorem 14.2.2. *Given enough training data, the θ maximizing (14.2) minimizes the KL divergence $KL(P||Q)$ where P is the true distribution and Q is the learnt distribution.*

Proof. This follows from

$$\begin{aligned} KL(P||Q) &= \mathop{\mathbb{E}}_{x\sim P}[\log \frac{P(x)}{Q(x)}] \\ &= \mathop{\mathbb{E}}_{x\sim P}[\log P(x)] - \mathop{\mathbb{E}}_{x\sim P}[\log Q(x)]. \end{aligned}$$

Notice that $\mathbb{E}_{x\sim P}[\log P(x)]$ is a constant that depends only upon the data distribution, and that computing log-likelihood using iid samples from P is like estimating the second term. We conclude that given enough samples, and a representation for Q that is expressive enough to capture P, minimizing $KL(P||Q)$ amounts to maximising log likelihood up to an additive constant. □

Note that except for low-dimensional settings, the previous Theorem does not give any meaningful bounds on the number of training datapoints needed for proper learning.

14.3 Energy-based Models: Conceptual Overview

AI and Machine Learning often use a powerful and ubiquitous idea from science and math: an energy landscape, with the system seeking to minimize its energy, settling in stable, low-energy valleys.

Energy-Based Models (EBMs) in machine learning leverage this exact physical intuition to model complex data, framing the entire learning problem as one of designing an energy landscape that is expressive enough and yet also allows efficient training.

The Core Idea: An Energy Function for Compatibility

Let's use a familiar, concrete example: generating images from text, or vice versa. Here, we have two types of data:

- **x**: An **image**, represented as a high-dimensional array of pixel values.
- **h**: A **text description** (terse or verbose) that corresponds to the image.

An EBM is defined by a single, scalar **energy function**, $E(x, h)$. This function takes an image x and a text description h and outputs a single number—the "energy"—which represents their incompatibility.

- **Low Energy:** $E(x, h)$ is low if the image x is a faithful and high-quality depiction of the text h. For example,

  $$E(\text{image\_of\_a\_cat, "a cat sitting on a mat"})$$

  would be very low.

- **High Energy:** $E(x, h)$ is high if the image and text are mismatched. $E(\text{image\_of\_a\_dog}, \text{"a cat sitting on a mat"})$ would be high. Likewise, a blurry, nonsensical image paired with any caption would also yield high energy.

The entire goal of training is to obtain a function E that assigns low energy to plausible, well-matched (x, h) pairs and high energy to everything else. Since we lack explicit mathematical description of E, it will be learned from data using a deep net.

Two Sides of the Coin: Generation and Interpretation

This simple framework, in principle, allows us to move between the image and text domains through optimization:

1. **Text-to-Image (Generation):** Given a fixed text description h_{fixed} (e.g., "An astronaut riding a horse on Mars"), the task is to find the image x that is most compatible. This becomes an optimization problem:

   $$x^* = \arg\min_x E(x, h_{\text{fixed}})$$

   In other words, we search through the vast space of all possible pixel combinations to find the one that creates a valley in our energy landscape for the given h.

2. **Image-to-Text (Captioning):** Given a fixed image x_{fixed}, we can find the most appropriate description h by minimizing the energy:

   $$h^* = \arg\min_h E(x_{\text{fixed}}, h)$$

   Here, we are searching through the space of all possible sentences to find the one that best describes the image.

This is an incredibly powerful and general framing. However, it runs into a massive computational problem: the partition function.

Bottleneck: Intractable Partition Function

To connect energy to the formal language of probability, we use the Gibbs-Boltzmann distribution:

$$P(x, h) = \frac{\exp(-E(x, h))}{Z}. \tag{14.3}$$

This defines the joint probability of a given (image, text) pair. The denominator, Z, is the infamous **partition function**, representing the sum over all possibilities to ensure the probabilities normalize to 1:

$$Z = \iint \exp(-E(x', h'))\, dx'\, dh'$$

Here is where we run into computational difficulty. To calculate Z, we would need to integrate over **every possible image** x' (an astronomically large space) and **every possible text string** h' (another immense, combinatorial space). This is fundamentally intractable. For decades, this has meant that EBMs, while theoretically elegant, were approximated in ways that made them impractical for complex, high-dimensional data.

Modern Breakthrough: Deep Learning Sidesteps the Problem

The generative AI revolution has in part relied on clever, non-trivial ways to use the energy function *without ever computing* Z. The key was to realize that for generation ($x^* = \arg\min_x E(x, h_{\text{fixed}})$), we don't need the absolute probabilities $P(x, h)$; we just need to know which way to go to find the energy minimum —specifically, the **gradient** of the energy landscape[3], which is captured by the *score function*. This insight gave rise to **Score-Based Models** and **Diffusion Models**, which are studied in Chapter 15.

[3] For example to find for a fixed x the minimizer h of $E(x, h)$ via gradient descent we only need $\nabla_h(E(x, h))$, and this is the score function for this setting.

14.4 An Older Idea: Variational method

(*This section is included for historical reasons. You can feel free to skip to Chapter 15 for more modern ideas in distributional modeling.*)

As sketched above, we are assuming a ground truth generative model $p(x, h)$ and we are assuming we have samples of x obtained by generating pairs of (x, h) according to the ground truth and discarding the h part. The *variational method* tries to learn $p(x)$ from such samples, where "variational"in the title refers to calculus of variations. It leverages *duality*, a widespread principle in math. The idea is to maintain a distribution $q(h|x)$ as an attempt to model $p(h|x)$ and improve a certain lower bound on $p(x)$. The key fact is the following. [4]

[4] See the blog post on offconvex.org by Arora and Risteski on how algorithms try to use some form of gradient descent or local improvement to improve $q(h|x)$.

Lemma 14.4.1 (ELBO Bound). *For any distribution* $q(h|x)$

$$\log p(x) \geq \mathbb{E}_{q(h|x)}[\log(p(x, h))] + H[q(h|x)], \tag{14.4}$$

where H *is the Shannon Entropy. (Note: equality is attained when* $q(h|x) = p(h|x)$*.)*

Proof. Since

$$KL[q(h|x) \,||\, p(h|x)] = \mathbb{E}_{q(h|x)}\left[\log \frac{q(h|x)}{p(h|x)}\right] \tag{14.5}$$

and $p(x)p(h|x) = p(x, h)$ (Bayes' Rule) we have:

$$KL[q(h|x)|p(h|x)] = \mathbb{E}_{q(h|x)}[\log \frac{q(h|x)}{p(x,h)} \cdot p(x)] \tag{14.6}$$

$$= \underbrace{\mathbb{E}_{q(h|x)}[\log(q(h|x))]}_{-H(q(h|x))} - \mathbb{E}_{q(h|x)}[\log(p(x,h))] + \mathbb{E}_{q(h|x)}[\log p(x)] \tag{14.7}$$

But since KL divergence is always nonnegative, so we get:

$$\mathbb{E}_{q(h|x)}[\log(p(x))] - \mathbb{E}_{q(h|x)}[\log(p(x,h))] - H(q(h|x)) \geq 0 \tag{14.8}$$

which leads to the desired inequality since $\log(p(x))$ is constant over $q(h|x)$ and thus $\mathbb{E}_{q(h|x)}[\log(p(x))] = p(x)$.

□

14.5 *Autoencoders and Variational Autoencoder (VAEs)*

(*This section is included for historical reasons. You can feel free to skip to Chapter 15 for more modern ideas in distributional modeling.*)

Autoencoders find a compressed latent representation h of the datapoint x such that x can be approximately recovered from h. They can be defined in multiple ways by chaging the formalization of what "approximate recovery" means.

In this section we formalize them using latent variable generative models. A popular instantiation of this in deep learning is *Variational Autoencoder (VAE)* [5]. As its name suggests two core classical ideas rest behind the design of VAEs: autoencoders – the original data $x \in \mathbb{R}^n$ is mapped into a high-level descriptor $z \in \mathbb{R}^d$ on a low dimensional (hopefully) meaningful manifold; variational inference – the objective to maximize is a lower bound on log-likelihood instead of the log-likelihood itself.

[5]

Recall that in density estimation we are given a data sample $x_1, \ldots, x_m$ and a parametric model $p_\theta(x)$, and our goal is to maximize the log-likelihood of the data: $\max_\theta \sum_{i=1}^m \log p_\theta(x_i)$. As a variational method, VAEs use the evidence lower bound (ELBO) as a training objective instead. For any distributions p on (x, z) and q on $z|x$, ELBO is derived from the fact that $KL(q(z|x) \,||\, p(z|x)) \geq 0$

$$\log p(x) \geq \mathbb{E}_{q(z|x)}[\log p(x,z)] - \mathbb{E}_{q(z|x)}[\log q(z|x)] = ELBO \tag{14.9}$$

where equality holds if and only if $q(z|x) \equiv p(z|x)$. In the VAE setting, the distribution $q(z|x)$ acts as the encoder, mapping a given data point x to a distribution of high-level descriptors, while $p(x,z) = p(z)p(x|z)$ acts as the decoder, reconstructing a distribution on data x

given a random seed $z \sim p(z)$. Deep learning comes in play for VAEs when constructing the aforementioned encoder q and decoder p. In particular,

$$q(z|x) = \mathcal{N}(z; \mu_x, \sigma_x^2 I_d), \quad \mu_x, \sigma_x = E_\phi(x) \tag{14.10}$$

$$p(x|z) = \mathcal{N}(x; \mu_z, \sigma_z^2 I_n), \quad \mu_z, \sigma_z = D_\theta(z), \quad p(z) = \mathcal{N}(z; 0, I_d) \tag{14.11}$$

where E_ϕ and D_θ are the encoder and decoder neural networks parameterized by ϕ and θ respectively, μ_x, μ_z are vectors of corresponding dimensions, and σ_x, σ_z are (nonnegative) scalars. The particular choice of Gaussians is not a necessity in itself for the model and can be replaced with any other relevant distribution. However, Gaussians provide, as is often the case, computational ease and intuitive backing. The intuitive argument behind the use of Gaussian distributions is that under mild regularity conditions every distribution can be approximated (in distribution) by a mixture of Gaussians. This follows from the fact that by approximating the CDF of a distribution by step functions one obtains an approximation in distribution by a mixture of constants, i.e. mixture of Gaussians with ≈ 0 variance. The computational ease, on the other hand, is more clearly seen in the training process of VAEs.

14.5.1 Training VAEs

As previously mentioned, the training of variational autoencoders involves maximizing the RHS of (14.9), the ELBO, over the parameters ϕ, θ under the model described by (14.10), (14.11). Given that the parametric model is based on two neural networks E_ϕ, D_θ, the objective optimization is done via gradient-based methods. Since the objective involves expectation over $q(z|x)$, computing an exact estimate of it, and consequently its gradient, is intractable so we resort to (unbiased) gradient estimators and eventually use a stochastic gradient-based optimization method (e.g. SGD).

In this section, use the notation $\mu_\phi(x), \sigma_\phi(x) = E_\phi(x)$ and $\mu_\theta(z), \sigma_\theta(z) = D_\theta(z)$ to emphasize the dependence on the parameters ϕ, θ. Given training data $x_1, \ldots, x_m \in \mathbb{R}^n$, consider an arbitrary data point $x_i, i \in [m]$ and pass it through the encoder neural network E_ϕ to obtain $\mu_\phi(x_i), \sigma_\phi(x_i)$. Next, sample s points $z_{i1}, \ldots, z_{is}$, where s is the batch size, from the distribution $q(z|x = x_i) = \mathcal{N}(z; \mu_\phi(x_i), \sigma_\phi(x_i)^2 I_d)$ via the reparameterization trick [6] by sampling $\epsilon_1, \ldots, \epsilon_s \sim \mathcal{N}(0, I_d)$ from the standard Gaussian and using the transformation $z_{ij} = \mu_\phi(x_i) + \sigma_\phi(x_i) \cdot \epsilon_j$. The reason behind the reparameterization trick is that the gradient w.r.t. parameter ϕ of an unbiased estimate of expectation over a general distribution q_ϕ is not necessarily an unbiased

[6]

estimate of the gradient of expectation. This is the case, however, when the distribution q_ϕ can separate the parameter ϕ from the randomness in the distribution, i.e. it's a deterministic transformation that depends on ϕ of a parameter-less distribution. With the s i.i.d. samples from $q(z|x = x_i)$ we obtain an unbiased estimate of the objective ELBO

$$\sum_{j=1}^{s} \log p(x_i, z_{ij}) - \sum_{j=1}^{s} \log q(z_{ij}|x_i) = \sum_{j=1}^{s} [\log p(x_i|z_{ij}) + \log p(z_{ij}) - \log q(z_{ij}|x_i)] \tag{14.12}$$

Here the batch size s indicates the fundamental tradeoff between computational efficiency and accuracy in estimation. Since each of the terms in the sum in (14.12) is a Gaussian distribution, we can write the ELBO estimate explicitly in terms of the parameter-dependent $\mu_\phi(x_i), \sigma_\phi(x_i), \mu_\theta(z_{ij}), \sigma_\theta(z_{ij})$ (while skipping some constants). A single term for $j \in [s]$ is given by

$$-\frac{1}{2}\left[\frac{||x_i - \mu_\theta(z_{ij})||^2}{\sigma_\theta(z_{ij})^2} + n \log \sigma_\theta(z_{ij})^2 + ||z_{ij}||^2 - \frac{||z_{ij} - \mu_\phi(x_i)||^2}{\sigma_\phi(x_i)^2} - d \log \sigma_\phi(x_i)^2\right] \tag{14.13}$$

Notice that (14.13) is differentiable with respect to all the components $\mu_\phi(x_i), \sigma_\phi(x_i), \mu_\theta(z_{ij}), \sigma_\theta(z_{ij})$ while each of these components, being an output of a neural network with parameters ϕ or θ, is differentiable with respect to the parameters ϕ or θ. Thus, the tractable gradient of the batch sum (14.12) w.r.t. ϕ (or θ) is, *due to the reparameterization trick*, an unbiased estimate of $\nabla_\phi ELBO$ (or $\nabla_\theta ELBO$) which can be used in any stochastic gradient-based optimization algorithm to maximize the objective ELBO and train the VAE.

15
Deep Generative Models

The fundamental goal of generative modeling is to learn a probability distribution $P_{data}(x)$ given only a finite set of samples $\{x_i\}$ drawn from it. A successful model should be able to perform two primary tasks:

1. **Sampling:** Generate new data points x_{new} from P_{data}, i.e., the same distribution as the training samples.
2. **Density Estimation:** Evaluate the probability density $P_{model}(x)$ for any given data point x.

As described in Chapter 14 the central challenge in probabilistic modeling is the intractability of the *partition function*, which makes direct computation of likelihood infeasible for flexible and expressive models. We also saw an old approach *Variational Autoencoders*. which avoids direct estimation of log likelihood by instead optimizing a lower bound to log likelihood. While early methods like Variational Autoencoders (VAEs) circumvented this by optimizing a surrogate objective (a lower bound), their sample quality often lagged and they have not scaled as effectively as a new generation of deep generative models. Today's AI derives its power from identifying general methods that continue to improve with more compute and data.

This chapter describes the modern revolution in deep generative models. We see several paradigms that find some clever ways to bypass the partition function problem.

15.1 The Landscape of Deep Generative Models

We organize the main strategies into three families, each defined by how it meets the challenge of representing and learning a complex distribution:

- **Explicit Density Models (The Path of Invertibility):** This approach defines a tractable formula for $P_{model}(x; \theta)$. The primary

example is **Normalizing Flows**, which use invertible transformations to map a simple base distribution to the complex data distribution, allowing for exact likelihood computation via the change of variables formula.

- **Score-Based and Diffusion Models (The Path of Gradients):** This third paradigm avoids modeling the density directly, and instead learns its gradient, the **score function**:

$$s_\theta(x) = \nabla_x \log P_\theta(x). \tag{15.1}$$

  Critically, the score is independent of the intractable partition function, since $\nabla_x \log P_\theta(x) = \nabla_x(-\log Z_\theta - E_\theta(x)) = -\nabla_x E_\theta(x)$. This insight allows for a new family of powerful generative models based on learning the geometry of the data distribution. We also cover the most general version of this idea, **Flow Matching**.

- **Implicit Density Models (The Path of Adversarial Games):** This idea predated the other methods, and a major set of developments involves **Generative Adversarial Networks (GANs)**. These provide a mechanism to sample from $P_{model}(x;\theta)$ without defining an explicit density function. The GAN framework can (with hindsight) be seen as an early attempt at learning a surrogate score function: the optimal discriminator learns a function of the density ratio $D^*(x) = \frac{P_{data}(x)}{P_{data}(x)+P_{model}(x)}$. This ratio is implicitly related to the difference in the log-densities (or scores), providing a signal to guide the generator. For more details see Chapter 17.

This chapter describes the key ideas of this modern revolution.

15.2 *Normalizing Flows*

The idea in *Normalizing Flows* (Rezende and Mohamed 2015) is to make the deep net *invertible*. Specifically, it computes a function $f_\theta : \Re^d \to \Re^d$ that is parametrized by trainable parameter vector θ and maps image x to its representation $h = f_\theta(x)$ (note: both have the same dimension). Importantly, f is an invertible map (i.e., one-to-one and onto) and differentiable (or almost everywhere differentiable). The advantage of such a transformation is that it gives a clear connection between the probability densities of x and h. In generative models h is assumed to have some prescribed probability density $\mu(h)$, usually uniform gaussian. Via the invertible map, this translates to a density $\rho(\cdot)$ on x given by

$$\rho(x) = \mu(f_\theta(x))|\det(J_f)| \tag{15.2}$$

where J_f is the Jacobian of f namely, whose (i, j) entry is $\partial f(x)_i / \partial x_j$ and $\det(\cdot)$ denotes determinant of the matrix. This exact expression for likelihood of the training datapoints allows usual gradient-based training.

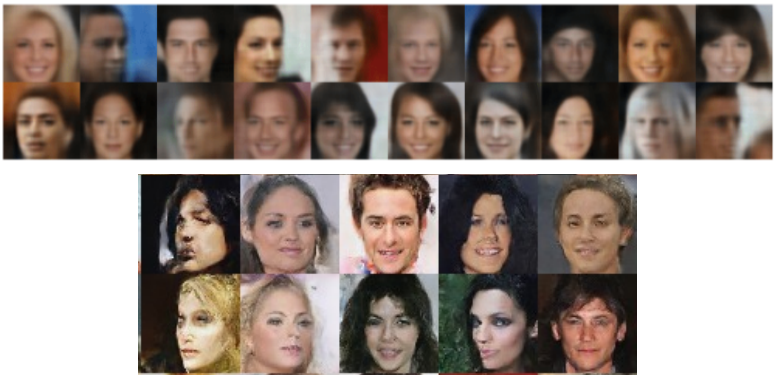

Figure 15.1: Faces in the top row were produced by a VAE based method and those in the second row by RealNVP using normalizing flows. VAE is known for producing blurry images. RealNVP's output is much better, but still has visible artifacts.

Which raises the question: how does one constrain nets to be invertible? Note that it suffices to constrain individual layers to be invertible, because the overall Jacobian is the composition of layer Jacobians. [1] To make layers invertible one often uses a variant of the following trick from the models NICE [2] and Real NVP [3]. If z^l is the input to layer l and z^{l+1} its output, then identify a special set of coordinates A in z^l and z^{l+1} and impose the restriction (where z_A denotes portion of z in the coordinates given by A, and B is shorthand for $\overline{A}$).

$$z_A^{l+1} = z_A^l \tag{15.3}$$
$$z_B^{l+1} = z_B^l \odot h_\theta(z_A^l) + s_\theta(z_A^l) \tag{15.4}$$

where $\odot$ denotes component-wise product and $h_\theta()$ is a function whose each output is nonnegative, with a convenient choice being to make it $\exp(r_\theta(z_A^l))$ for some other function $r_\theta()$.

This layer is invertible because given z^{l+1} one can recover z^l as follows:

$$z_A^l = z_A^{l+1} \tag{15.5}$$
$$z_B^l = (z_B^{l+1} - s_\theta(z_A^l)) \odot h_\theta(z_A^l) \tag{15.6}$$

Note that the choice of A, B can change from layer to layer, so all coordinates may get updated as they go through multiple layers. Furthermore, denoting by $z|_A$ the portion of the layer vector on coordinates A, the Jabobian for the layer mapping is lower triangular. Hence the determinant is the product of the diagonal entries. [4]

$$\frac{\partial}{\partial z^l} z^{l+1} = \begin{pmatrix} I_{|A|\times|A|} & 0 \\ \frac{\partial}{z^l|_A} z^{l+1}|_B & \text{diag}(h_\theta(z_A^l)) \end{pmatrix}$$

[1] Since $\det(AB) = \det(A)\det(B)$ the determinant of the deep net's Jacobian is the product of the determinants of the layers.

[2] Laurent Dinh, David Krueger, and Yoshua Bengio. NICE: Nonlinear Independent Component Analysis. *Proc. ICLR*, 2015

[3] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density Eestimation using Real NVP. *Proc. ICLR*, 2017

[4] Recall that the training loss ultimately uses log probabilities, i.e., log of the expression in (15.2). Since the determinant is product of diagonal entries, after taking logs we obtain the sum of log of diagonal entries. This results in nice clean expressions for the gradient.

Normalizing flows can be extended to convolutional nets by restricting the convolutions to be 1×1. Then convolutional filters just involve scalings of channel values, and the corresponding Jacobian is a diagonal nonzero matrix. Also the split of coordinates into A and B split can be done within channels as well. This is one of the ideas in GLOW model [5], which can generate better images than its predecessors.

[5] Diederik P. Kingma and Prafulla Dhariwal. GLOW: Generative Flow with Invertible 1×1 convolutions. *Proc. Neurips*, 2019

Some auto-regressive models such as PixelCNN are capable of producing very realistic-looking images from random seeds. However, they do not fit into the distribution learning paradigm described above so we do not discuss them here. They involve generating the image pixel by pixel (roughly speaking) and thus do not parallelize well.

Problem 15.2.1. *Let (z_1, z_2, z_3, z_4) be distributed as a standard Gaussian $\mathcal{N}(0, I)$ in $\mathbb{R}^4$. Let $f : \mathbb{R}^4 \to \mathbb{R}^4$ be an invertible function which maps (z_1, z_2, z_3, z_4) to $(z_1, z_2, e^{a_0} z_3 + a_1 z_1^2 + a_2 z_2^2, e^{b_0} z_4 + b_1 z_1^2 + b_2 z_2^2)$ for some coefficients $a_0, a_1, a_2, b_0, b_1, b_2 \in \mathbb{R}$. Compute the probability density function of $f(z_1, z_2, z_3, z_4)$.*

15.3 *Diffusion Models*

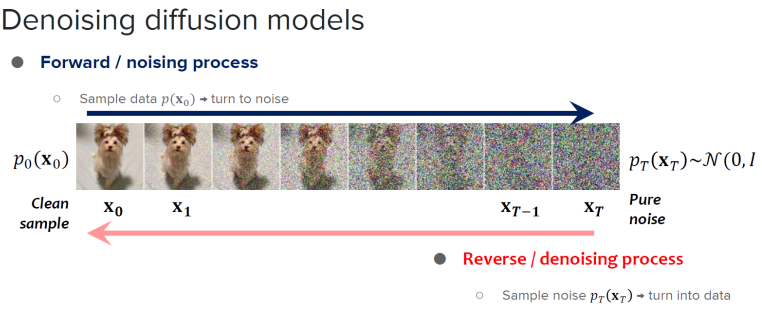

Figure 15.2: Example of noising an image and then denoising, using Diffusion Model. (Source: Binxu Wang)

You may have seen AI models that generate artificial imagery given a text prompt such as "Penguin walking in an orange Princeton jacket." These are made by *diffusion models* [6]. They are reminiscent of normalizing flows and autoencoders, in that they define a mapping f that transforms the set of all images to a sample from $\mathcal{N}(0, I)$, as well as an inverse mapping f^{-1} that maps vectors from the normal distribution to images. The new angle here is that f is very trivial; just a series of noising steps. In other words, f progressively destroys structure in data by adding noise. Then f^{-1} is a net that is custom-trained for denoising the output of f.

[6] J-Sohl-Dickstein, E. Weiss, N Maheshwaranathan, S. Ganguli. *Deep Unsupervised Learning using Nonequilibrium Thermodynamics. 2015*

15.3.1 The Forward Process: Progressive Noising

The "forward process" is a fixed procedure that gradually transforms a data point $x_0 \sim P_{data}$ into pure isotropic Gaussian noise over a series of T discrete timesteps.

It is a Markov chain[7] defined by:

$$q(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t I). \tag{15.7}$$

[7] Markov Chain is a probabilistic process that is "history-less;" meaning the next step only depends on the current state/location, and not the history that led to it.

Here, $\{\beta_t\}_{t=1}^T$ is a pre-defined variance schedule, where the β_t are small positive constants. A key property is that we can sample x_t directly from x_0. Letting $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{i=1}^t \alpha_i$ it can be shown that:

$$q(x_t|x_0) = \mathcal{N}(x_t; \sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t)I). \tag{15.8}$$

If the schedule is chosen such that $\bar{\alpha}_T \approx 0$, then x_T is distributed as a standard Gaussian, $\mathcal{N}(0, I)$, irrespective of the starting point x_0. Typically the schedule stops just short of this point, so that information about the original x_0 is retained.

Problem 15.3.1. *Prove the assertion in (15.8). You should need the fact that if z_1, z_2 are samples from $\mathcal{N}(\mu, I)$, then $\alpha z_1 + \beta z_2$ is a sample from $\mathcal{N}((\alpha+\beta)\mu, (\alpha^2+\beta^2)I)$*

15.3.2 The Reverse Process: Denoising, Scores, and Guidance

The model's generative power comes from learning the *reverse process*: starting with the highly noised image $x_T \sim \mathcal{N}(0, I)$ and iteratively denoising it to produce a sample x_0.

Why is such a denoising feasible? This is feasible only if the forward noising steps are small enough to be statistically reversible. A high-dimensional perspective clarifies why a deep network can learn this reversal.

Consider a noisy vector $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon$, where $x_0 \in \mathbb{R}^n$ is the image (signal) and $\epsilon \sim \mathcal{N}(0, I)$ is noise. While samples from $\mathcal{N}(0, I)$ have an ℓ_2-norm concentrated around $\sqrt{n}$, making the noise component's magnitude much larger than the signal's, the signal's directionality remains detectable. A pure noise vector is isotropic; its projection on any fixed direction is a zero-mean Gaussian. The vector x_t, however, has a non-zero mean projection onto the direction of x_0. A sufficiently powerful network can learn to detect this anisotropic structure, distinguishing it from pure noise.

This learning is achieved by training a neural network, $\epsilon_\theta(x_t, t)$, to predict the noise component ϵ from the noisy input x_t. The training objective is a simple mean-squared error loss:

$$\mathcal{L}(\theta) = \mathbb{E}_{x_0,\epsilon,t}\left[\|\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, t)\|^2\right]. \tag{15.9}$$

This objective forces the network to learn the statistical properties of the signal manifold (i.e., real images) to effectively isolate it from the noise.

This denoising task has a deep connection to the geometry of the data distribution. The objective is an instance of **Denoising Score Matching**. The optimal network, ϵ_θ^*, learns a function directly proportional to the **score**, which is the gradient of the log-probability of the noisy data distribution:

$$\epsilon_\theta^*(x_t, t) \propto \nabla_{x_t} \log p(x_t). \tag{15.10}$$

Thus, by learning to predict a direction that reduces the noise (i.e., slightly denoises), the network implicitly learns a vector field that points from any noisy point x_t back towards the manifold of higher data density. One would apply this network repeatedly to denoise further, finally ending up at a realistic image, which is also a local maximum of the log-probability (and hence the above gradient is zero).[8] In a very real sense, repeatedly applying the network on the noised image is a form of gradient ascent!

[8]

For **conditional generation**, the network is augmented to accept a condition c (e.g., a text embedding), becoming $\epsilon_\theta(x_t, t, c)$. It is trained on the same objective, but a fraction of the time c is replaced with a null token $\emptyset$. This forces the network to learn both the conditional score $\nabla_{x_t} \log p(x_t|c)$ and the unconditional score $\nabla_{x_t} \log p(x_t)$ with the same weights. At inference, **Classifier-Free Guidance (CFG)** uses both predictions to amplify the condition. The final noise estimate is an extrapolation away from the unconditional prediction:

$$\widehat{\epsilon} = \epsilon_\theta(x_t, t, \emptyset) + w \cdot (\epsilon_\theta(x_t, t, c) - \epsilon_\theta(x_t, t, \emptyset)). \tag{15.11}$$

The guidance scale $w > 1$ strengthens the influence of the condition c, pushing the sample to be a more faithful representation of the prompt.

15.4 Flow Matching: A Deterministic Path from Noise to Data

Diffusion models learn to reverse a complex, stochastic noising process. This can be visualized as learning to trace winding, random paths from a simple noise distribution (a Gaussian) back to the complex data manifold. Flow Matching offers a more direct approach: can we learn a deterministic, straight-line transformation from noise to data?

The goal is to learn a time-dependent **velocity vector field** $v_\theta(z, t)$, parameterized by a neural network. This field defines a "flow" that transports samples from a noise distribution P_0 (e.g., $\mathcal{N}(0, I)$) at $t = 0$

to the data distribution P_1 at $t = 1$. The path of any sample z_t is governed by an Ordinary Differential Equation (ODE):

$$\frac{dz_t}{dt} = v_\theta(z_t, t). \tag{15.12}$$

Generating a sample means starting with $z_0 \sim P_0$ and solving this ODE from $t = 0$ to $t = 1$.

The challenge is to train $v_\theta(z, t)$ without the high cost of solving an ODE at every training step. Flow Matching's insight is to use a simple, elegant proxy task based on the "straight-line" intuition. The training procedure is as follows:

1. **Pair Samples:** Draw one sample from the noise distribution, $z_0 \sim P_0$, and one from the data, $z_1 \sim P_1$.

2. **Define a Straight Path:** Construct a linear interpolation between them: $z_t = (1 - t)z_0 + tz_1$.

3. **Define a Target Velocity:** The velocity required to traverse this specific path is constant and trivial: $v_{\text{target}} = z_1 - z_0$.

4. **Train the Network:** For a random time t and the corresponding point z_t on the path (a "tick mark"), train the network to predict this target velocity. The loss is a simple regression objective:

$$\mathcal{L}(\theta) = \mathbb{E}_{z_0, z_1, t}\left[\|v_\theta((1 - t)z_0 + tz_1, t) - (z_1 - z_0)\|^2\right]. \tag{15.13}$$

The network does not merely memorize these straight paths. By training on millions of such examples, it is forced to learn a single, globally consistent vector field that effectively "stitches together" the local velocity information from all the individual paths. This process compels the model to learn the underlying structure of the data manifold.

At inference, generation is a deterministic, step-by-step process. We start with a single random noise vector, $z^{(0)} \sim P_0$. This vector is then progressively transformed over N discrete steps to create the final image. For each step i from 0 to $N - 1$:

1. We calculate the current time fraction, $t = i/N$.

2. We feed the current vector $z^{(i)}$ and time t into the network to get a velocity vector: $v = v_\theta(z^{(i)}, t)$.

3. We update our vector by taking a small, straight step in this direction:

$$z^{(i+1)} = z^{(i)} + \frac{1}{N} \cdot v_\theta(z^{(i)}, i/N). \tag{15.14}$$

After N steps, the final vector $z^{(N)}$ is the generated sample. This process traces a smooth, deterministic trajectory from the noise distribution to the data manifold. Because the path is direct and contains no random noise injection at each step (unlike in diffusion models), it can often be approximated accurately with far fewer steps, enabling significantly faster generation.

Diffusion Models vs Flow Matching From our discussion it should be clear that Diffusion Models can be seen as a special case of Flow Matching albeit using a different sets of paths between the image manifold and $N(O, I)$. [9].

[9] For more on this see:
R. Gao, E. Hoogeboom, J. Heek, J. Heek1, V. De Bortoli, KP Murphy, T. Salimans. *Diffusion Models and Gaussian Flow Matching: Two Sides of the Same Coin.*
`https://diffusionflow.github.io/`

15.5 *Physics Simulation by learning Vector Fields*

The principles of Flow Matching (specifically, as a method to learn vector fields from trajectories) are related to a similar powerful framework for design of deep models for prediction of phenomena in natural systems.

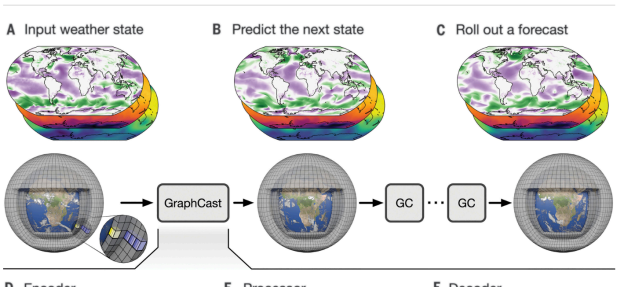

Figure 15.3: Weather prediction with GraphCast model. (Illustration taken from the paper.)

To illustrate we consider prediction of weather dynamics, exemplified by the famous GraphCast model [10]. In this scientific application, the goal is not to learn a flow from random noise to a valid data point (like an image), but rather to learn the flow from one valid data point to the next one in a time-series. The "datapoint" is current state S_t of a weather system (example: weather system over the Gulf of Mexico during hurricane season) and the desired path corresponds to evolution of the global weather state, S_t, over time according to laws of physics. GraphCast's objective is to learn the time-dependent **velocity vector field** $v(S, t)$ that defines this physical

[10] R. Lam et al. *Learning skillful medium-range global weather forecasting* **Science**, 2023.

flow. Crucially, the model isn't trained on sparse, raw observations. It uses decades of "re-analysis" data (like the ERA5 dataset), where a traditional physics-based simulation on a supercomputer is continuously corrected with real-world measurements. This provides a complete, consistent, and physically plausible history of global weather, creating a dense dataset suitable for training a large neural network.

The training process directly implements the core idea of learning vector fields. For any given state S_t from the historical data, the network is trained to predict the "target velocity," which in this context is the observed change to the next state, $\Delta S = S_{t+\Delta t} - S_t$. This is conceptually similar to the denoising process in diffusion models, where the network $\epsilon_\theta(x_t, t)$ learns to predict the update required to move from a noisy state to a cleaner one. To generate a long-range forecast, GraphCast applies this learned update rule autoregressively: it takes the current weather S_t, predicts the state 6 hours into the future, and then uses that new predicted state as the input to predict the state 12 hours out, and so on. In this way, it effectively simulates the trajectory of the weather forward in time by repeatedly applying the data-driven dynamics it learned from history.

Chapter Problems

Problem 15.5.1 (Normalizing Flow Jacobian)**.** *Consider a 2-dimensional Normalizing Flow layer that maps an input $z = (z_1, z_2)$ to an output $x = (x_1, x_2)$ using the following RealNVP-style transformation:*

$$\begin{aligned} x_1 &= z_1 \\ x_2 &= z_2 \cdot \exp(s(z_1)) + t(z_1) \end{aligned}$$

where $s(z_1) = \alpha z_1^2$ and $t(z_1) = \beta z_1$ for scalar constants α, β. Assume the input z is drawn from a standard 2D Gaussian, $p_Z(z) = \mathcal{N}(z; 0, I)$. [11]

[11] The expression $\mathcal{N}(z; 0, I)$ should be read as ""The distribution of z is a Gaussian of mean 0 and covariance matrix I.

(i) Find the inverse transformation that maps x back to z.

(ii) Compute the Jacobian matrix of the forward transformation, $J = \frac{\partial x}{\partial z}$.

(iii) Show that the determinant of the Jacobian is $\exp(s(z_1))$.

(iv) Using the change of variables formula, write down the probability density function $p_X(x)$ for the output vector x.

Problem 15.5.2 (Diffusion Forward Process)**.** *The forward diffusion process is defined by the Markov chain $q(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t I)$. The goal is to prove the closed-form sampling equation for any timestep t.*

Let $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$. Using induction, prove that:

$$q(x_t|x_0) = \mathcal{N}(x_t; \sqrt{\bar{\alpha}_t}x_0, (1 - \bar{\alpha}_t)I)$$

12

Problem 15.5.3 (Diffusion Reverse Posterior). *A key insight in diffusion models is that the objective of predicting the noise ϵ is equivalent to predicting the mean of the reverse process posterior distribution, $q(x_{t-1}|x_t, x_0)$. Your goal is to derive this connection.*

(i) Using Bayes' theorem for probability densities, $p(a|b,c) \propto p(b|a,c)p(a|c)$, show that the reverse posterior can be written as:

$$q(x_{t-1}|x_t, x_0) \propto \exp\left(-\frac{1}{2}\left(\frac{(x_t - \sqrt{\alpha_t}x_{t-1})^2}{\beta_t} + \frac{(x_{t-1} - \sqrt{\bar{\alpha}_{t-1}}x_0)^2}{1-\bar{\alpha}_{t-1}}\right)\right)$$

13

(ii) The posterior $q(x_{t-1}|x_t, x_0)$ is known to be a Gaussian. By completing the square for the terms involving x_{t-1} in the exponent, derive its mean, $\widetilde{\mu}_t(x_t, x_0)$, and show that it can be expressed as:

$$\widetilde{\mu}_t(x_t, x_0) = \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}x_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}x_t$$

(iii) Finally, substitute $x_0 = \frac{1}{\sqrt{\bar{\alpha}_t}}(x_t - \sqrt{1-\bar{\alpha}_t}\epsilon)$ into the expression for the mean. Show that this reparameterizes the mean as a function of x_t and ϵ. This proves that a network trained to predict ϵ from x_t can be used to construct the mean of the reverse step distribution.

Problem 15.5.4 (Flow Matching Target Velocity). *In the Flow Matching framework, the network input is $z_t = (1-t)z_0 + tz_1$ and the target velocity is $v = z_1 - z_0$.*

(i) Show that for any $t \in (0,1)$, the target vector v can be expressed in terms of the input z_t and either endpoint:

$$v = \frac{z_t - z_0}{t} = \frac{z_1 - z_t}{1-t}$$

(ii) Explain the implication of this result. Why does this property ensure that the learning task for the velocity field $v_\theta(z_t, t)$ is well-posed and not arbitrary, even though z_0 and z_1 are paired randomly?

[12] Hint: Recall that if $z_1 \sim \mathcal{N}(\mu_1, \sigma_1^2 I)$ and $z_2 \sim \mathcal{N}(\mu_2, \sigma_2^2 I)$ are independent, then $az_1 + bz_2 \sim \mathcal{N}(a\mu_1 + b\mu_2, (a^2\sigma_1^2 + b^2\sigma_2^2)I)$.

[13] Hint: $q(x_{t-1}|x_t, x_0)$ is proportional to $q(x_t|x_{t-1}, x_0)q(x_{t-1}|x_0)$. Note that the first term simplifies because the process is Markovian.