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

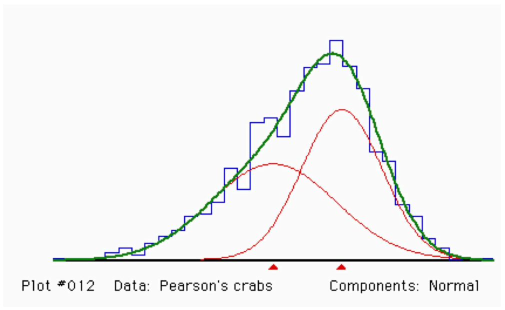

Figure 14.1: Visualization of Pearson's Crab Data as mixture of two Gaussians. (Credit: MIX homepage at McMaster University.)

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

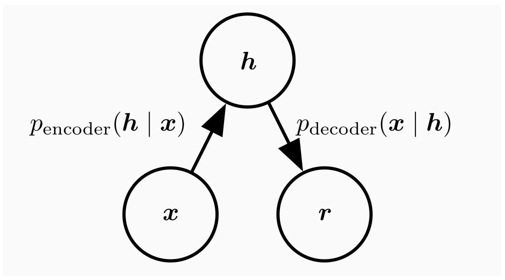
h
$p_{\text{encoder}}(h \mid x)$
$p_{\text{decoder}}(x \mid h)$
x
r

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(image_of_a_dog, "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 $\mathbb{E}_{q(h|x)}[\log(p(x))] = p(x)$.

$\square$

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.

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 $f_\theta \colon \Re^d \to \Re^d$ 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.

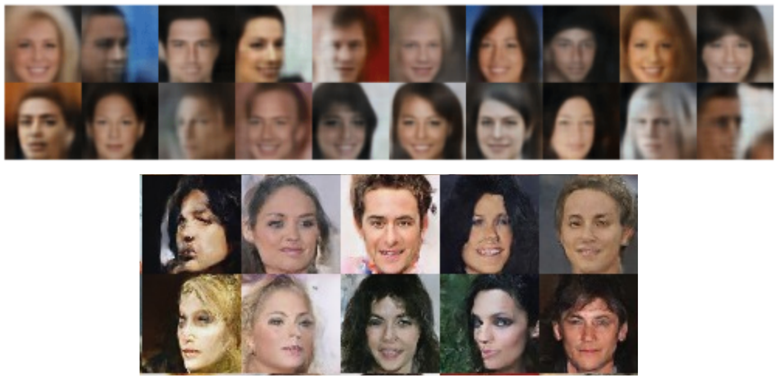
Figure 15.1: Faces in the top row were produced by a VAE based method and those in the second row by RealNVP using normalizing flows. VAE is known for producing blurry images. RealNVP's output is much better, but still has visible artifacts.

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 $\overline{A}$).

[1] Since $\det(AB) = \det(A)\mathbf{det}(B)$ 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

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

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]

[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.

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

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*

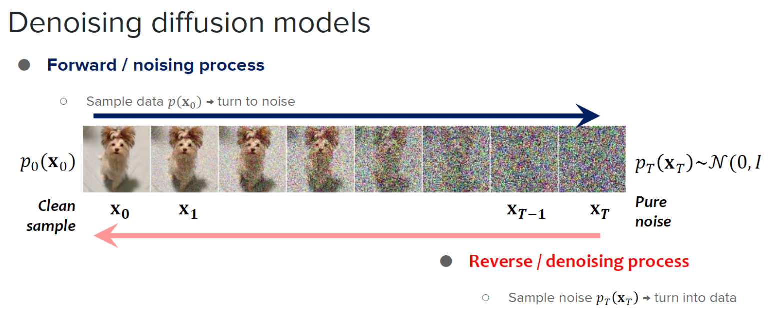
Denoising diffusion models
Forward / noising process
Sample data $p(\mathbf{x}_0)$ → turn to noise
$p_0(\mathbf{x}_0)$
$p_T(\mathbf{x}_T) \sim \mathcal{N}(0, I$
Clean sample
$\mathbf{x}_0$
$\mathbf{x}_1$
$\mathbf{x}_{T-1}$
$\mathbf{x}_T$
Pure noise
Reverse / denoising process
Sample noise $p_T(\mathbf{x}_T)$ → turn into data

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 and the Intuition of Reversibility*

The generative power of the model comes from learning the *reverse process*: starting with noise $x_T \sim \mathcal{N}(0, I)$ and iteratively denoising it to produce a sample x_0. This requires modeling $p_\theta(x_{t-1}|x_t)$. For this to be feasible, the forward process must be reversible in a statistical sense; that is, the information about x_{t-1} must not be completely erased in x_t. This is only possible if the noise steps β_t are small.

Let's build intuition in 1-D. The basic fact is that $1D$ gaussians with the same standard deviation σ become easily distinguishable once the inter-center distance t is a small multiple of σ

Problem 15.3.2 (Distinguishability of 1D Gaussians)**.** *Consider two univariate Gaussian distributions, $P = \mathcal{N}(\mu_1, \sigma^2)$ and $Q = \mathcal{N}(\mu_2, \sigma^2)$, with the same variance σ^2. Let the distance between their means be $t = |\mu_1 - \mu_2|$. A simple classifier assigns a given sample x to the distribution whose mean is closer. Assume that a sample is drawn with equal probability from either P or Q.*

(i) Show that the total probability of this classifier making an error is given by $1 - \Phi\left(\frac{t}{2\sigma}\right)$, where $\Phi(\cdot)$ is the cumulative distribution function (CDF) of the standard normal distribution $\mathcal{N}(0,1)$. [8]

[8] Hint: Without loss of generality, set the means to $-t/2$ and $+t/2$ and find the decision boundary. Then calculate the probability of a sample from one of the Gaussians falling on the wrong side of this boundary.

(ii) Using the standard tail bound for a Gaussian, $1 - \Phi(z) \leq \frac{1}{2}e^{-z^2/2}$ for $z > 0$, show that the error probability decays exponentially with the

square of the ratio (t/σ). Specifically, show that $P(error) \leq \frac{1}{2}\exp\left(-\frac{t^2}{8\sigma^2}\right)$. σ. (iii) Generalize the calculation to gaussian in d dimensions. Show that the same classifier continues to work for the same σ. (Note: but σ^2 is variance per direction. So the "radius" of the gaussian is $\sigma\sqrt{d}$!)

Consider two distinct starting points, x_0 and x_0'. After one step of noising, they become distributions $q(x_1|x_0) = \mathcal{N}(\sqrt{\alpha_1}x_0, \beta_1)$ and $q(x_1|x_0') = \mathcal{N}(\sqrt{\alpha_1}x_0', \beta_1)$. We can reliably distinguish samples from these two Gaussians if the distance between their centers is larger than their standard deviation. The distance between centers is $\sqrt{\alpha_1}|x_0 - x_0'|$, and the standard deviation is $\sqrt{\beta_1}$. The process remains largely invertible if:

$$\sqrt{\alpha_1}|x_0 - x_0'| \gg \sqrt{\beta_1} \implies |x_0 - x_0'| \gg \sqrt{\frac{\beta_1}{1-\beta_1}}. \tag{15.9}$$

For small β_1, this means the noise added at each step, $\sqrt{\beta_1}$, must be significantly smaller than the typical distances between distinct data points. By keeping β_t small, we ensure that the distributions $q(x_t|x_{t-1})$ corresponding to different previous states x_{t-1} have significant overlap. In other words the information about the original image x_0 degrades gradually and this makes a meaningful reverse process possible.

Because the forward steps are small, the reverse transition $p_\theta(x_{t-1}|x_t)$ can also be approximated by a Gaussian. The central challenge is to estimate the mean of this Gaussian. If we had access to the original x_0, the reverse posterior $q(x_{t-1}|x_t, x_0)$ would be tractable. Since we don't, we train a neural network to predict the necessary information from x_t. It turns out that this is equivalent to training the network to predict the noise component ϵ that was added to create x_t.

Let our model be a neural network $\epsilon_\theta(x_t, t)$. The training objective is remarkably simple:

1. Draw a real data sample $x_0 \sim P_{data}$ and a noise sample $\epsilon \sim \mathcal{N}(0, I)$.
2. Choose a random timestep $t \in \{1, \dots, T\}$.
3. Create the noisy sample $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon$.
4. Update the network parameters θ by taking a gradient step on the loss[9]:

$$\mathcal{L}(\theta) = \mathbb{E}_{x_0,\epsilon,t}[\|\epsilon - \epsilon_\theta(x_t, t)\|^2]. \tag{15.10}$$

[9] Note that this gradient step helps the deep net learn properties of the "signal" in the ϵ which has to be used for producing its estimate ϵ_θ. See Problem 15.3.3.

Once trained, we generate samples by starting with $x_T \sim \mathcal{N}(0, I)$ and iterating backwards from $t = T$ down to 1:

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(x_t, t)\right) + \sigma_t z, \quad \text{where } z \sim \mathcal{N}(0, I). \tag{15.11}$$

Problem 15.3.3. *(i) Using Bayes' theorem for Gaussians, derive the expression for the mean and variance of the true posterior $q(x_{t-1}|x_t, x_0)$. (ii) Show that parameterizing the reverse process mean $\widetilde{\mu}_t(x_t, x_0)$ is equivalent to predicting the noise ϵ from x_t.*

15.3.3 Putting it all together: Conditional Generation

The process described above generates unconditional samples; it learns the overall distribution $P(x)$ and can draw random samples from it. However, the true power of modern diffusion models is revealed in their ability to perform **conditional generation**—creating an image x given some input c, thus learning $P(x|c)$. This input c can be a class label, a semantic embedding of a text prompt, or even another image.

The most effective and widely used technique for this is known as **Classifier-Free Guidance (CFG)**. It cleverly enables conditional generation without requiring a separate classifier network. The idea is to train a single diffusion model that can operate in both a conditional and an unconditional mode.

Training. The denoising network is augmented to accept a conditioning vector, becoming $\epsilon_\theta(x_t, t, c)$. During training, with a certain fixed probability (e.g., 10-20% of the time), the conditioning vector c is replaced with a special null token, $\emptyset$. This forces the network to learn to predict the noise both with and without guidance from the same set of weights.

Sampling. During sampling, this dual capability is exploited to steer the generation process. At each step t, the model makes two forward passes to predict the noise:

- A conditional prediction: $\epsilon_{cond} = \epsilon_\theta(x_t, t, c)$
- An unconditional prediction: $\epsilon_{uncond} = \epsilon_\theta(x_t, t, \emptyset)$

The final noise prediction used for the denoising step is then an extrapolation away from the unconditional prediction in the direction of the conditional one:

$$\widehat{\epsilon} = \epsilon_{uncond} + w \cdot (\epsilon_{cond} - \epsilon_{uncond}) \tag{15.12}$$

Here, w is the *guidance scale*, a scalar hyperparameter typically set to a value greater than 1 (e.g., 7.5).

The intuition is powerful: the difference vector $(\epsilon_{cond} - \epsilon_{uncond})$ represents the direction in which the denoising process should move to better conform to the condition c. By setting $w > 1$, we amplify this direction, pushing the sample to be a more faithful and often higher-quality example of the conditioned concept, sometimes at the expense of sample diversity. This simple but highly effective trick is a cornerstone of modern high-fidelity text-to-image models.

The process described above generates unconditional samples; it learns the overall distribution $P(x)$ and can draw random samples from it. However, the true power of modern diffusion models is revealed in their ability to perform **conditional generation**—creating an image x given some input c, thus learning $P(x|c)$. This input c can be a class label, a semantic embedding of a text prompt, or even another image.

15.3.4 The Deeper Connection: Denoising is Score Matching

The remarkably simple noise-prediction objective has a beautiful theoretical justification. What our network is implicitly learning is the **score function** of the noisy data distribution, $s(x_t) = \nabla_{x_t} \log p(x_t)$. The score function points in the direction of steepest ascent for the log-probability density.

It can be shown that the optimal noise-predicting function, $\epsilon^*(x_t, t)$, is directly related to the score of the marginal distribution $p(x_t)$:

$$\epsilon^*(x_t, t) = -\sqrt{1-\bar{\alpha}_t}\nabla_{x_t} \log p(x_t). \tag{15.13}$$

Therefore, by training our network ϵ_θ to predict the noise ϵ using a simple MSE loss, we are implicitly training it to compute the score of the data distribution at every noise level t.

This perspective, known as **Denoising Score Matching**, explains why the diffusion process works so well. A major challenge of learning the score for the true data distribution $p(x_0)$ is that the function can be very complex, and data is sparse in many regions of the space. By contrast, the noisy distributions $p(x_t)$ for $t > 0$ are smoother and cover the entire space, making their scores easier to learn. The diffusion model elegantly learns the scores for this whole continuum of distributions and uses them to guide samples from pure noise back to the complex data manifold.

15.4 Flow Matching: A Unifying Perspective

The success of diffusion models, understood through the lens of score matching, raises a profound question: are the complex, stochastic paths of the diffusion process necessary? Or can we learn to transform noise into data more directly? This leads to the elegant and unifying framework of **Flow Matching**.

15.4.1 The Goal: Learning a Direct Transformation

The core objective is to learn a direct mapping from a simple base distribution P_0 (e.g., a standard Gaussian $\mathcal{N}(0, I)$) to the complex data distribution P_1 (P_{data}). A naive attempt to train a network f by minimizing $\mathbb{E}_{z_0 \sim P_0, z_1 \sim P_1}[|f(z_0) - z_1|^2]$ is ill-posed, as the random pairing of a specific noise vector z_0 to a specific data point z_1 is arbitrary.

Flow matching solves this by reframing the problem. Instead of learning a direct map, we learn a time-dependent **velocity vector field** $v(z, t)$, parameterized by a neural network. This vector field defines a "flow" that transports samples from P_0 at $t = 0$ to P_1 at $t = 1$. The path of any particle z_t is governed by an Ordinary Differential Equation (ODE):

$$\frac{dz_t}{dt} = v(z_t, t). \tag{15.14}$$

Generating a sample z_1 then corresponds to starting with $z_0 \sim P_0$ and solving this ODE from $t = 0$ to $t = 1$.

15.4.2 Simple solution: Training on Straight Lines

The central challenge is how to train the vector field $v(z, t)$ without solving a costly ODE at every training step. The insight of Flow Matching is to train the network on a simple, well-defined proxy task. The procedure is as follows:

1. **Pair Samples:** Draw a random noise sample $z_0 \sim P_0$ and a real data sample $z_1 \sim P_1$.

2. **Define a Simple Path:** Construct a straight-line path between these points: $z_t = (1 - t)z_0 + tz_1$.

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

4. **Train the Network:** Train the network $v(z, t)$ using a simple regression objective. For a random time t and the corresponding point z_t on the path, the loss is:

$$\mathcal{L}(\theta) = \mathbb{E}_{z_0, z_1, t}[|v(z_t, t; \theta) - (z_1 - z_0)|^2]. \tag{15.15}$$

This objective is non-arbitrary because, for the given context of a path from a specific z_0 to a specific z_1, the target velocity is uniquely defined. By training on millions of such straight-line examples, the network is forced to learn a globally consistent vector field that smoothly interpolates between them.

15.5 Geometric Intuition: Flow-matching vs Diffusion

We point out how Diffusion Models are a special case of Flow Matching[10]

[10] 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/`

15.5.1 Comparing the training method

The training objective in both cases involves using datapoints each of which is a noised version of a randomly chosen image, and the deep net has to output a small correction that slightly denoises it.

For diffusion, we saw that even when a signal vector is corrupted by high-dimensional Gaussian noise of a much larger magnitude, the signal's original direction remains clearly detectable via projection. Flow Matching relies on a cleaner, non-stochastic version of this same principle of a persistent signal.

Problem 15.5.1 (Information Preservation in Flows)**.** *In the Flow Matching setup, the network input is $z_t = (1-t)z_0 + tz_1$ and the target velocity is $v = z_1 - z_0$. Show that the target vector v can be expressed in terms of the input z_t and the endpoints:*

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

This shows that for any $t \in (0,1)$, the target velocity is always a re-scaling of the vector pointing from the start of the path to the current position, and from the current position to the end of the path. The directional signal is never lost.

This persistent signal makes the learning task well-posed. However, generalization does not come from the model memorizing these straight-line paths. Instead, the network is forced to learn a single, continuous vector field $v(z,t)$ that must simultaneously satisfy millions of these simple, local constraints. The network must effectively "stitch together" or average all of these local, straight-line instructions into a globally consistent flow field. This process forces the model to learn the underlying structure of the data manifold.

This learning task has a natural duality. When t is close to 1, the input z_t is a slightly perturbed data point, and the task is essentially denoising, forcing the model to learn fine details. When t is close to 0, the input is nearly pure noise, and the task is to impose coarse, global structure. By learning a single field that handles all t, the network is trained to understand the data distribution at all scales, which is the foundation of its powerful generative capability.

But in both cases modeling can be seen as learning a vector field that induces a probability flow from noise to data. The two methods differ in specific choice of paths—stochastic and wiggly in one case and deterministic and straight in the other.

15.5.2 *Comparison of the generative process*

The training method is very different but once the models are trained, the generative process for both Diffusion and Flow Matching models follows similar iterative structure: start with a noise vector, and progressively refine it over N steps by applying a learned neural network. However, the specific update equations reveal that they are simulating fundamentally different mathematical processes.

Diffusion Models: Forward process via SDE solver
The standard algorithm for generating a sample from a diffusion model (specifically, a Denoising Diffusion Probabilistic Model or DDPM) involves reversing the noising process. This is mathematically equivalent to numerically solving a Stochastic Differential Equation (SDE).

The sampling algorithm proceeds as follows, counting down from time T to 0:

1. **Initialize:** Draw a single sample from the base distribution, $x_T \sim \mathcal{N}(0, I)$.
2. **Iterate:** For $t = T, T-1, \ldots, 1$, compute the next state x_{t-1} from the current state x_t:
   - Evaluate the trained neural network to predict the noise component from the current state:
     $$\epsilon_{\text{pred}} = \epsilon_\theta(x_t, t)$$
   - Apply the reverse update rule. This involves a deterministic "denoising" step and a stochastic "re-noising" step:
     $$x_{t-1} = \underbrace{\frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(x_t, t)\right)}_{\text{Deterministic Component}} + \underbrace{\sigma_t z}_{\text{Stochastic Component}}, \quad \text{where } z \sim \mathcal{N}(0, I). \tag{15.16}$$

     The term σ_t is a non-zero variance (for $t > 1$) that re-injects noise at each step.
3. **Final Sample:** The generated sample is the final state after T steps, $x_{new} = x_0$.

Flow Matching: Solving an Ordinary Differential Equation (ODE)
As derived previously, the Flow Matching sampler is a numerical ODE solver. Its structure is parallel to the diffusion sampler, but its update rule is purely deterministic.

1. **Initialize:** Draw a single sample, $z^{(0)} \sim \mathcal{N}(0, I)$.
2. **Iterate:** For $i = 0, 1, \ldots, N-1$, compute the next state $z^{(i+1)}$:

$$z^{(i+1)} = z^{(i)} + \frac{1}{N} \cdot v(z^{(i)}, i/N; \theta) \tag{15.17}$$

3. **Final Sample:** The generated sample is the final state, $x_{new} = z^{(N)}$.

The Crucial Difference. Comparing Equation 15.16 and Equation 15.17 clarifies the core distinction. While both use a learned network (ϵ_θ vs. v_θ) to update a sample iteratively, the diffusion update includes the explicit stochastic term $\sigma_t z$. This term is the mathematical signature of an SDE solver, forcing the generative path to be noisy and complex, thus requiring a large number of steps (T) for an accurate solution. The Flow Matching update has no such term. It is a deterministic ODE solver, tracing a smooth path that can be accurately approximated with a much smaller number of steps (N), leading to significantly faster generation.

15.6 Learning Physics as Vector Fields

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

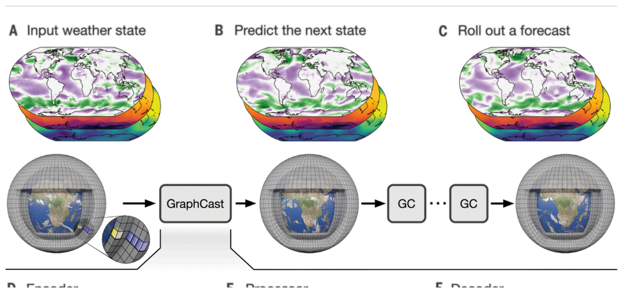

Figure 15.3: Weather prediction with GraphCast model. (Illustration taken from the paper.)

To illustrate we consider prediction of weather dynamics, exemplified by the famous GraphCast model [11]. 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

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

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 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.