Part IV

Generation

20 Generative models: an overview

20.1 Introduction

A **generative model** is a joint probability distribution $p(\boldsymbol{x})$, for $\boldsymbol{x} \in \mathcal{X}$. In some cases, the model may be conditioned on inputs or covariates $\boldsymbol{c} \in \mathcal{C}$, which gives rise to a **conditional generative model** of the form $p(\boldsymbol{x}|\boldsymbol{c})$.

There are many kinds of generative model. We give a brief summary in Section 20.2, and go into more detail in subsequent chapters. See also [Tom22] for a recent book on this topic that goes into more depth.

20.2 Types of generative model

There are many kinds of generative model, some of which we list in Table 20.1. At a high level, we can distinguish between **deep generative models** (DGM) — which use deep neural network to learn a complex mapping from a single latent vector $\boldsymbol{z}$ to the observed data $\boldsymbol{x}$ — and more "classical" **probabilistic graphical models** (PGM), that map a set of interconnected latent variables $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_L$ to the observed variables $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_D$ using simpler, often linear, mappings. Of course, many hybrids are possible. For example, PGMs can use neural networks, and DGMs can use structured state spaces. We discuss PGMs in general terms in Chapter 4, and give examples in Chapter 28, Chapter 29, Chapter 30. In this part of the book, we mostly focus on DGMs.

The main kinds of DGM are: **variational autoencoders** (**VAE**), **autoregressive models** (**ARM**) models, **normalizing flows**, **diffusion models**, **energy based models** (**EBM**), and **generative adversarial networks** (**GAN**). We can categorize these models in terms of the following criteria (see Figure 20.1 for a visual summary):

- Density: does the model support pointwise evaluation of the probability density function $p(\boldsymbol{x})$, and if so, is this fast or slow, exact, approximate or a bound, etc? For **implicit models**, such as GANs, there is no well-defined density $p(\boldsymbol{x})$. For other models, we can only compute a lower bound on the density (VAEs), or an approximation to the density (EBMs, UPGMs).
- Sampling: does the model support generating new samples, $\boldsymbol{x} \sim p(\boldsymbol{x})$, and if so, is this fast or slow, exact or approximate? Directed PGMs, VAEs, and GANs all support fast sampling. However, undirected PGMs, EBMs, ARM, diffusion, and flows are slow for sampling.
- Training: what kind of method is used for parameter estimation? For some models (such as AR, flows and directed PGMs), we can perform exact maximum likelihood estimation (MLE), although

Model	Chapter	Density	Sampling	Training	Latents	Architecture
PGM-D	Section 4.2	Exact, fast	Fast	MLE	Optional	Sparse DAG
PGM-U	Section 4.3	Approx, slow	Slow	MLE-A	Optional	Sparse graph
VAE	Chapter 21	LB, fast	Fast	MLE-LB	$\mathbb{R}^L$	Encoder-Decoder
ARM	Chapter 22	Exact, fast	Slow	MLE	None	Sequential
Flows	Chapter 23	Exact, slow/fast	Slow	MLE	$\mathbb{R}^D$	Invertible
EBM	Chapter 24	Approx, slow	Slow	MLE-A	Optional	Discriminative
Diffusion	Chapter 25	LB	Slow	MLE-LB	$\mathbb{R}^D$	Encoder-Decoder
GAN	Chapter 26	NA	Fast	Min-max	$\mathbb{R}^L$	Generator-Discriminator

Table 20.1: Characteristics of common kinds of generative model. Here D is the dimensionality of the observed $\boldsymbol{x}$, and L is the dimensionality of the latent $\boldsymbol{z}$, if present. (We usually assume $L \ll D$, although overcomplete representations can have $L \gg D$.) Abbreviations: Approx = approximate, ARM = autoregressive model, EBM = energy based model, GAN = generative adversarial network, MLE = maximum likelihood estimation, MLE-A = MLE (approximate), MLE-LB = MLE (lower bound), NA = not available, PGM = probabilistic graphical model, PGM-D = directed PGM, PGM-U = undirected PGM, VAE = variational autoencoder.

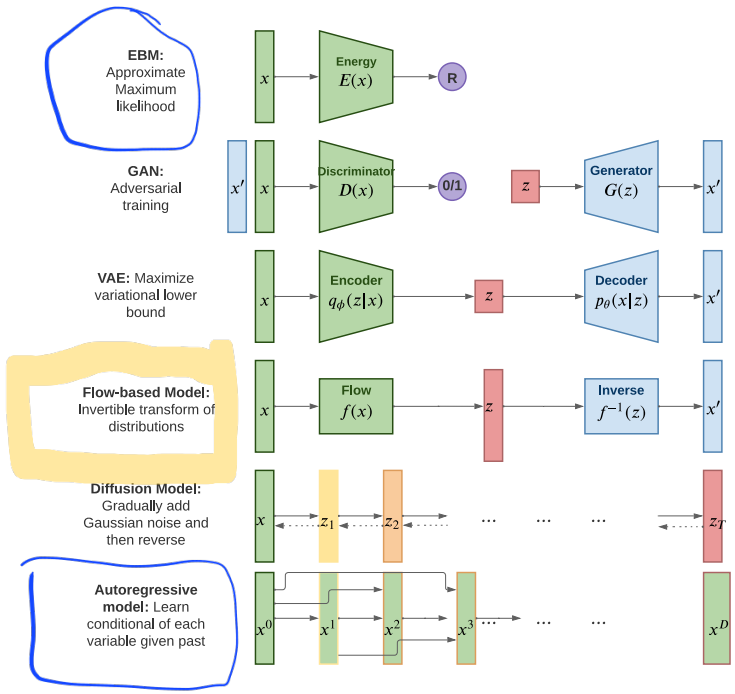


Figure 20.1: Summary of various kinds of deep generative model. Here $\boldsymbol{x}$ is the observed data, $\boldsymbol{z}$ is the latent code, and $\boldsymbol{x}'$ is a sample from the model. AR models do not have a latent code $\boldsymbol{z}$. For diffusion models and flow models, the size of $\boldsymbol{z}$ is the same as $\boldsymbol{x}$. For AR models, x^d is the d'th dimension of $\boldsymbol{x}$. R represents real-valued output, 0/1 represents binary output. Adapted from Figure 1 of [Wen21].

the objective is usually non-convex, so we can only reach a local optimum. For other models, we cannot tractably compute the likelihood. In the case of VAEs, we maximize a lower bound on the likelihood; in the case of EBMs and UGMs, we maximize an approximation to the likelihood. For GANs we have to use min-max training, which can be unstable, and there is no clear objective function to monitor.

- Latents: does the model use a latent vector $\boldsymbol{z}$ to generate $\boldsymbol{x}$ or not, and if so, is it the same size as $\boldsymbol{x}$ or is it a potentially compressed representation? For example, ARMs do not use latents; flows and diffusion use latents, but they are not compressed.[1] Graphical models, including EBMs, may or may not use latents.
- Architecture: what kind of neural network should we use, and are there restrictions? For flows, we are restricted to using invertible neural networks where each layer has a tractable Jacobian. For EBMs, we can use any model we like. The other models have different restrictions.

20.3 Goals of generative modeling

There are several different kinds of tasks that we can use generative models for, as we discuss below.

20.3.1 Generating data

One of the main goals of generative models is to generate (create) new data samples. This is sometimes called **generative AI** (see e.g., [GBGM23] for a recent survey). For example, if we fit a model $p(\boldsymbol{x})$ to images of faces, we can sample new faces from it, as illustrated in Figure 25.10.[2] Similar methods can be used to create samples of text, audio, etc. When this technology is abused to make fake content, they are called **deep fakes** (see e.g., [Ngu+19]). Genrative models can also be used to create **synthetic data** for training discriminative models (see e.g., [Wil+20; Jor+22]).

To control what is generated, it is useful to use a **conditional generative model** of the form $p(\boldsymbol{x}|\boldsymbol{c})$. Here are some examples:

- $\boldsymbol{c}$ = text prompt, $\boldsymbol{x}$ = image. This is a **text-to-image** model (see Figure 20.2, Figure 20.3 and Figure 22.6 for examples).
- $\boldsymbol{c}$ = image, $\boldsymbol{x}$ = text. This is an **image-to-text** model, which is useful for **image captioning**.
- $\boldsymbol{c}$ = image, $\boldsymbol{x}$ = image. This is an **image-to-image** model, and can be used for image colorization, inpainting, uncropping, JPEG artefact restoration, etc. See Figure 20.4 for examples.
- $\boldsymbol{c}$ = sequence of sounds, $\boldsymbol{x}$ = sequence of words. This is a **speech-to-text** model, which is useful for **automatic speech recognition** (**ASR**).
- $\boldsymbol{c}$ = sequence of English words, $\boldsymbol{x}$ = sequence of French words. This is a **sequence-to-sequence** model, which is useful for **machine translation**.

1. Flow models define a latent vector $\boldsymbol{z}$ that has the same size as $\boldsymbol{x}$, although the internal deterministic computation may use vectors that are larger or smaller than the input (see e.g., the DenseFlow paper [GGS21]).
2. These images were made with a technique called score-based generative modeling (Section 25.3), although similar results can be obtained using many other techniques. See for example https://this-person-does-not-exist.com/en which shows results from a GAN model (Chapter 26).

(a) Teddy bears swimming at the Olympics 400m Butterfly event.

(b) A cute corgi lives in a house made out of sushi.

(c) A cute sloth holding a small treasure chest. A bright golden glow is coming from the chest.

Figure 20.2: Some 1024×1024 *images generated from text prompts by the Imagen diffusion model (Section 25.6.4). From Figure 1 of [Sah+22b]. Used with kind permission of William Chan.*

Figure 20.3: Some images generated from the Parti transformer model (Section 22.4.2) in response to a text prompt. We show results from models of increasing size (350M, 750M, 3B, 20B). Multiple samples are generated, and the highest ranked one is shown. From Figure 10 of [Yu+22]. Used with kind permission of Jiahui Yu.

- $\boldsymbol{c}$ = initial prompt, $\boldsymbol{x}$ = continuation of the text. This is another sequence-to-sequence model, which is useful for automatic **text generation** (see Figure 22.5 for an example).

Note that, in the conditional case, we sometimes denote the inputs by $\boldsymbol{x}$ and the outputs by $\boldsymbol{y}$. In this case the model has the familiar form $p(\boldsymbol{y}|\boldsymbol{x})$. In the special case that $\boldsymbol{y}$ denotes a low dimensional quantity, such as a integer class label, $y \in \{1, \ldots, C\}$, we get a predictive (discriminative) model. The main difference beween a discriminative model and a conditional generative model is this: in a discriminative model, we assume there is one correct output, whereas in a conditional generative model, we assume there may be multiple correct outputs. This makes it harder to evaluate generative models, as we discuss in Section 20.4.

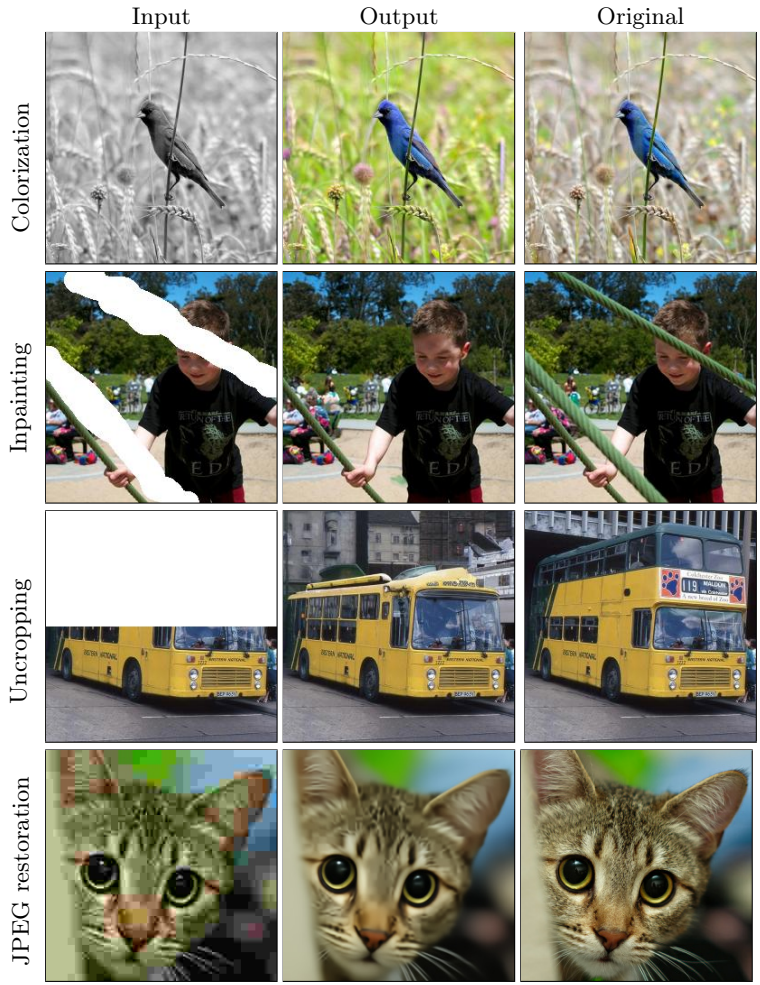


Figure 20.4: Illustration of some image-to-image tasks using the Palette conditional diffusion model (Section 25.6.4). From Figure 1 of [Sah+22a]. Used with kind permission of Chitwan Saharia.

20.3.2 Density estimation

The task of **density estimation** refers to evaluating the probablity of an observed data vector, i.e., computing $p(\boldsymbol{x})$. This can be useful for outlier detection (Section 19.3.2), data compression (Section 5.4), generative classifiers, model comparison, etc.

A simple approach to this problem, which works in low dimensions, is to use **kernel density**

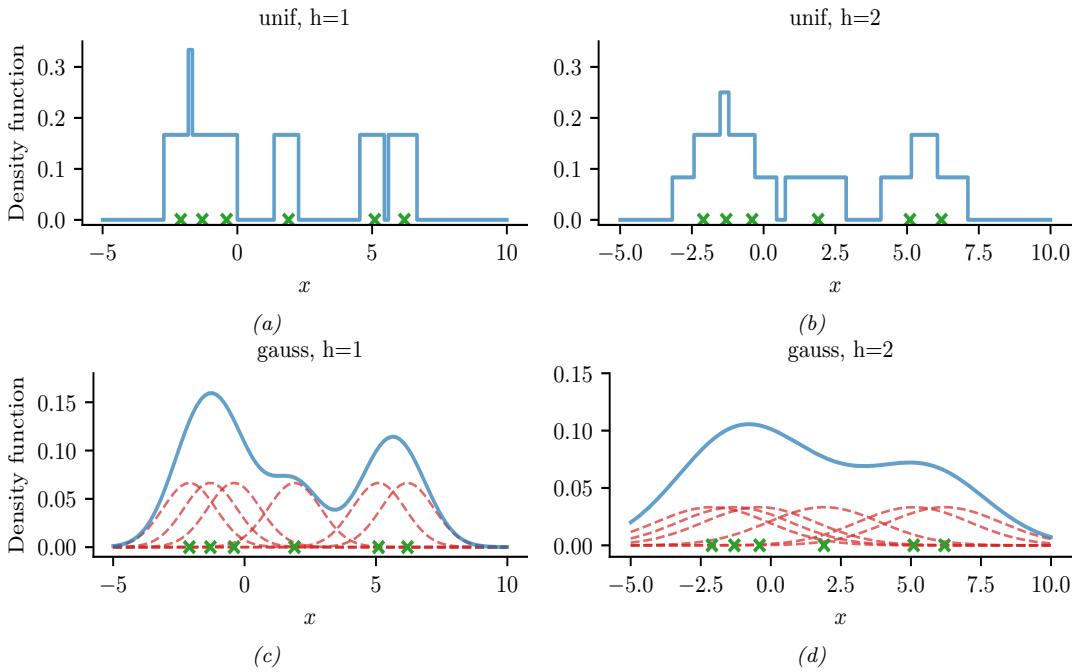


Figure 20.5: A nonparametric (Parzen) density estimator in 1d estimated from 6 datapoints, denoted by x. Top row: uniform kernel. Bottom row: Gaussian kernel. Left column: bandwidth parameter $h = 1$. Right column: bandwidth parameter $h = 2$. Adapted from http://en.wikipedia.org/wiki/Kernel_density_estimation*. Generated by parzen_window_demo.ipynb.*

estimation or **KDE**, which has the form

$$p(\boldsymbol{x}|\mathcal{D}) = \frac{1}{N}\sum_{n=1}^{N} \mathcal{K}_h\left(\boldsymbol{x} - \boldsymbol{x}_n\right) \tag{20.1}$$

Here $\mathcal{D} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$ is the data, and $\mathcal{K}_h$ is a density kernel with **bandwidth** h, which is a function $\mathcal{K} : \mathbb{R} \to \mathbb{R}_+$ such that $\int \mathcal{K}(x)dx = 1$ and $\int x\mathcal{K}(x)dx = 0$. We give a 1d example of this in Figure 20.5: in the top row, we use a uniform (boxcar) kernel, and in the bottom row, we use a Gaussian kernel.

In higher dimensions, KDE suffers from the **curse of dimensionality** (see e.g., [AHK01]), and we need to use parametric density models $p_{\boldsymbol{\theta}}(\boldsymbol{x})$ of some kind.

20.3.3 Imputation

The task of **imputation** refers to "filling in" missing values of a data vector or data matrix. For example, suppose $\mathbf{X}$ is an $N \times D$ matrix of data (think of a spreadsheet) in which some entries, call them $\mathbf{X}_m$, may be missing, while the rest, $\mathbf{X}_o$, are observed. A simple way to fill in the missing data is to use the mean value of each feature, $\mathbb{E}[x_d]$; this is called **mean value imputation**, and is

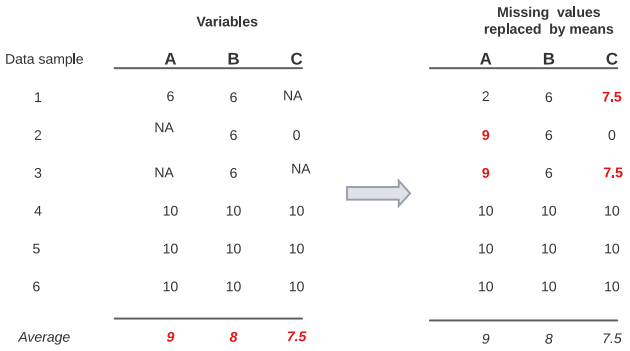


Figure 20.6: Missing data imputation using the mean of each column.

illustrated in Figure 20.6. However, this ignores dependencies between the variables within each row, and does not return any measure of uncertainty.

We can generalize this by fitting a generative model to the observed data, $p(\mathbf{X}_o)$, and then computing samples from $p(\mathbf{X}_m|\mathbf{X}_o)$. This is called **multiple imputation**. A generative model can be used to fill in more complex data types, such as **in-painting** occluded pixels in an image (see Figure 20.4).

See Section 3.11 for a more general discussion of missing data.

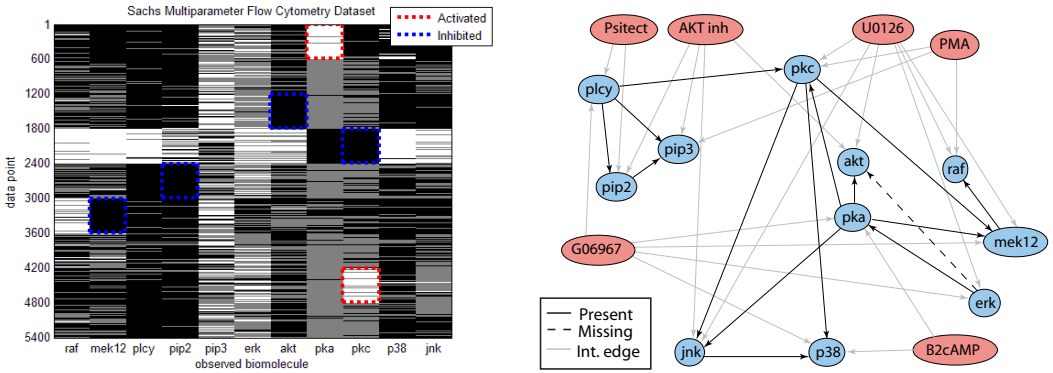
20.3.4 Structure discovery

Some kinds of generative models have latent variables $\boldsymbol{z}$, which are assumed to be the "causes" that generated the observed data $\boldsymbol{x}$. We can use Bayes' rule to invert the model to compute $p(\boldsymbol{z}|\boldsymbol{x}) \propto p(\boldsymbol{z})p(\boldsymbol{x}|\boldsymbol{z})$. This can be useful for discovering latent, low-dimensional patterns in the data.

For example, suppose we perturb various proteins in a cell and measure the resulting phosphorylation state using a technique known as flow cytometry, as in [Sac+05]. An example of such a dataset is shown in Figure 20.7(a). Each row represents a data sample $\boldsymbol{x}_n \sim p(\cdot|\boldsymbol{a}_n, \boldsymbol{z})$, where $\boldsymbol{x} \in \mathbb{R}^{11}$ is a vector of outputs (phosphorylations), $\boldsymbol{a} \in \{0,1\}^6$ is a vector of input actions (perturbations) and $\boldsymbol{z}$ is the unknown cellular signaling network structure. We can infer the graph structure $p(\boldsymbol{z}|\mathcal{D})$ using graphical model structure learning techhniques (see Section 30.3). In particular, we can use the dynamic programming method described in [EM07] to get the result is shown in Figure 20.7(b). Here we plot the median graph, which includes all edges for which $p(z_{ij} = 1|\mathcal{D}) > 0.5$. (For a more recent approach to this problem, see e.g., [Bro+20b].)

20.3.5 Latent space interpolation

One of the most interesting abilities of certain latent variable models is the ability to generate samples that have certain desired properties by interpolating between existing datapoints in latent space. To explain how this works, let $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ be two inputs (e.g., images), and let $\boldsymbol{z}_1 = e(\boldsymbol{x}_1)$ and $\boldsymbol{z}_2 = e(\boldsymbol{x}_2)$ be their latent encodings. (The method used for computing these will depend on the



(a) *(b)*

Figure 20.7: (a) A design matrix consisting of 5400 datapoints (rows) measuring the state (using flow cytometry) of 11 proteins (columns) under different experimental conditions. The data has been discretized into 3 states: low (black), medium (grey), and high (white). Some proteins were explicitly controlled using activating or inhibiting chemicals. (b) A directed graphical model representing dependencies between various proteins (blue circles) and various experimental interventions (pink ovals), which was inferred from this data. We plot all edges for which $p(G_{ij} = 1|\mathcal{D}) > 0.5$. Dotted edges are believed to exist in nature but were not discovered by the algorithm (1 false negative). Solid edges are true positives. The light colored edges represent the effects of intervention. From Figure 6d of [EM07].

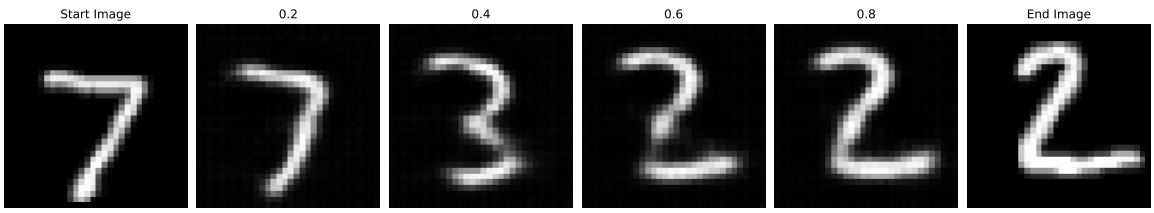


Figure 20.8: Interpolation between two MNIST images in the latent space of a β-VAE (with $\beta = 0.5$). Generated by mnist_vae_ae_comparison.ipynb.

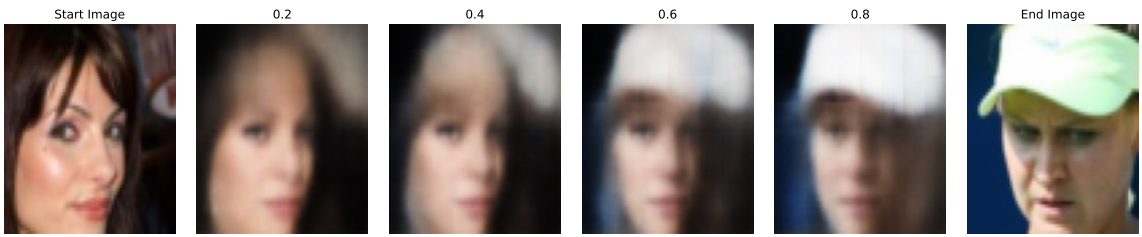


Figure 20.9: Interpolation between two CelebA images in the latent space of a β-VAE (with $\beta = 0.5$). Generated by celeba_vae_ae_comparison.ipynb.

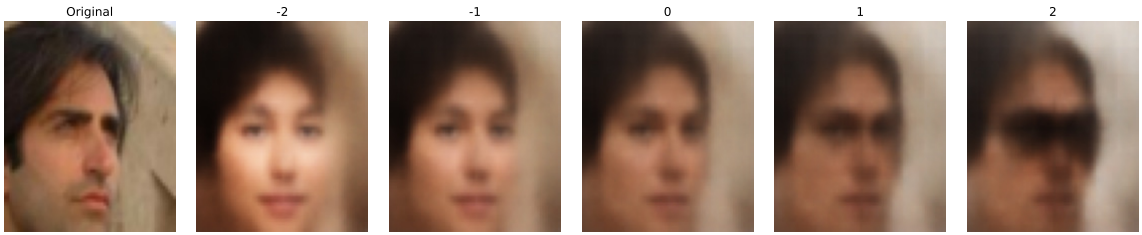


Figure 20.10: Arithmetic in the latent space of a β-VAE (with $\beta = 0.5$). The first column is an input image, with embedding $\boldsymbol{z}$. Subsequent columns show the decoding of $\boldsymbol{z} + s\boldsymbol{\Delta}$, where $s \in \{-2, -1, 0, 1, 2\}$ and $\boldsymbol{\Delta} = \overline{\boldsymbol{z}}^+ - \overline{\boldsymbol{z}}^-$ is the difference in the average embeddings of images with or without a certain attribute (here, wearing sunglasses). Generated by celeba_vae_ae_comparison.ipynb.

type of model; we discuss the details in later chapters.) We can regard $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$ as two "anchors" in latent space. We can now generate new images that interpolate between these points by computing $\boldsymbol{z} = \lambda \boldsymbol{z}_1 + (1 - \lambda)\boldsymbol{z}_2$, where $0 \leq \lambda \leq 1$, and then decoding by computing $\boldsymbol{x}' = d(\boldsymbol{z})$, where $d()$ is the decoder. This is called **latent space interpolation**, and will generate data that combines semantic features from both $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$. (The justification for taking a linear interpolation is that the learned manifold often has approximately zero curvature, as shown in [SKTF18]. However, sometimes it is better to use nonlinear interpolation [Whi16; MB21; Fad+20].)

We can see an example of this process in Figure 20.8, where we use a β-VAE model (Section 21.3.1) fit to the MNIST dataset. We see that the model is able to produce plausible interpolations between the digit 7 and the digit 2. As a more interesting example, we can fit a β-VAE to the **CelebA** dataset [Liu+15].[3] The results are shown in Figure 20.9, and look reasonable. (We can get much better quality if we use a larger model trained on more data for a longer amount of time.)

It is also possible to perform interpolation in the latent space of text models, as illustrated in Figure 21.7.

20.3.6 Latent space arithmetic

In some cases, we can go beyond interpolation, and can perform **latent space arithmetic**, in which we can increase or decrease the amount of a desired "semantic factor of variation". This was first shown in the **word2vec** model [Mik+13], but it also is possible in other latent variable models. For example, consider our VAE model fit to the CelebA dataset, which has faces of celebrities and some corresponding attributes. Let $\mathbf{X}_i^+$ be a set of images which have attribute i, and $\mathbf{X}_i^-$ be a set of images which do not have this attribute. Let $\mathbf{Z}_i^+$ and $\mathbf{Z}_i^-$ be the corresponding embeddings, and $\overline{\boldsymbol{z}}_i^+$ and $\overline{\boldsymbol{z}}_i^-$ be the average of these embeddings. We define the offset vector as $\boldsymbol{\Delta}_i = \overline{\boldsymbol{z}}_i^+ - \overline{\boldsymbol{z}}_i^-$. If we add some positive multiple of $\boldsymbol{\Delta}_i$ to a new point $\boldsymbol{z}$, we increase the amount of the attribute i; if we subtract some multiple of $\boldsymbol{\Delta}_i$, we decrease the amount of the attribute i [Whi16].

We give an example of this in Figure 20.10. We consider the attribute of wearing sunglasses. The j'th reconstruction is computed using $\hat{\boldsymbol{x}}_j = d(\boldsymbol{z} + s_j \boldsymbol{\Delta})$, where $\boldsymbol{z} = e(\boldsymbol{x})$ is the encoding of the original image, and s_j is a scale factor. When $s_j > 0$ we add sunglasses to the face. When $s_j < 0$ we

3. CelebA contains about 200k images of famous celebrities. The images are also annotated with 40 attributes. We reduce the resolution of the images to 64×64, as is conventional.

remove sunglasses; but this also has the side effect of making the face look younger and more female, possibly a result of dataset bias.

20.3.7 Generative design

Another interesting use case for (deep) generative models is **generative design**, in which we use the model to generate candidate objects, such as molecules, which have desired properties (see e.g., [RNA22]). One approach is to fit a VAE to unlabeled samples, and then to perform Bayesian optimization (Section 6.6) in its latent space, as discussed in Section 21.3.5.2.

20.3.8 Model-based reinforcement learning

We discuss reinforcement learning (RL) in Chapter 35. The main success stories of RL to date have been in computer games, where simulators exist and data is abundant. However, in other areas, such as robotics, data is expensive to acquire. In this case, it can be useful to learn a generative "**world model**", so the agent can do planning and learning "in it's head". See Section 35.4 for more details.

20.3.9 Representation learning

Representation learning refers to learning (possibly uninterpretable) latent factors $\boldsymbol{z}$ that generate the observed data $\boldsymbol{x}$. The primary goal is for these features to be used in "**downstream**" supervised tasks. This is discussed in Chapter 32.

20.3.10 Data compression

Models which can assign high probability to frequently occuring data vectors (e.g., images, sentences), and low probability to rare vectors, can be used for **data compression**, since we can assign shorter codes to the more common items. Indeed, the optimal coding length for a vector $\boldsymbol{x}$ from some stochastic source $p(\boldsymbol{x})$ is $l(\boldsymbol{x}) = -\log p(\boldsymbol{x})$, as proved by Shannon. See Section 5.4 for details.

20.4 Evaluating generative models

This section is written by Mihaela Rosca, Shakir Mohamed, and Balaji Lakshminarayanan.

Evaluating generative models requires metrics which capture

- **sample quality** — are samples generated by the model a part of the data distribution?
- **sample diversity** — are samples from the model distribution capturing all modes of the data distribution?
- **generalization** — is the model generalizing beyond the training data?

There is no known metric which meets all these requirements, but various metrics have been proposed to capture different aspects of the learned distribution, some of which we discuss below.

20.4.1 Likelihood-based evaluation

A standard way to measure how close a model q is to a true distribution p is in terms of the KL divergence (Section 5.1):

$$D_{\mathbb{KL}}\left(p \parallel q\right) = \int p(\boldsymbol{x}) \log \frac{p(\boldsymbol{x})}{q(\boldsymbol{x})} = -\mathbb{H}\left(p\right) + \mathbb{H}_{ce}\left(p, q\right) \tag{20.2}$$

where $\mathbb{H}\left(p\right)$ is a constant, and $\mathbb{H}_{ce}\left(p, q\right)$ is the cross entropy. If we approximate $p(\boldsymbol{x})$ by the empirical distribution, we can evaluate the cross entropy in terms of the empirical **negative log likelihood** on the dataset:

$$\text{NLL} = -\frac{1}{N}\sum_{n=1}^{N} \log q(\boldsymbol{x}_n) \tag{20.3}$$

Usually we care about negative log likelihood on a held-out test set.[4]

20.4.1.1 Computing the log-likelihood

For models of discrete data, such as language models, it is easy to compute the (negative) log likelihood. However, it is common to measure performance using a quantity called **perplexity**, which is defined as 2^H, where $H = \text{NLL}$ is the cross entropy or negative log likelihood.

For image and audio models, one complication is that the model is usually a continuous distribution $p(\boldsymbol{x}) \geq 0$ but the data is usually discrete (e.g., $\boldsymbol{x} \in \{0, \ldots, 255\}^D$ if we use one byte per pixel). Consequently the average log likelihood can be arbitrary large, since the pdf can be bigger than 1. To avoid this it is standard pratice to use **uniform dequantization** [TOB16], in which we add uniform random noise to the discrete data, and then treat it as continuous-valued data. This gives a lower bound on the average log likelihood of the discrete model on the original data.

To see this, let $\boldsymbol{z}$ be a continuous latent variable, and $\boldsymbol{x}$ be a vector of binary observations computed by rounding, so $p(\boldsymbol{x}|\boldsymbol{z}) = \delta(\boldsymbol{x} - \text{round}(\boldsymbol{z}))$, computed elementwise. We have $p(\boldsymbol{x}) = \int p(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})d\boldsymbol{z}$. Let $q(\boldsymbol{z}|\boldsymbol{x})$ be a probabilistic inverse of $\boldsymbol{x}$, that is, it has support only on values where $p(\boldsymbol{x}|\boldsymbol{z}) = 1$. In this case, Jensen's inequality gives

$$\log p(\boldsymbol{x}) \geq \mathbb{E}_{q(\boldsymbol{z}|\boldsymbol{x})}\left[\log p(\boldsymbol{x}|\boldsymbol{z}) + \log p(\boldsymbol{z}) - \log q(\boldsymbol{z}|\boldsymbol{x})\right] \tag{20.4}$$

$$= \mathbb{E}_{q(\boldsymbol{z}|\boldsymbol{x})}\left[\log p(\boldsymbol{z}) - \log q(\boldsymbol{z}|\boldsymbol{x})\right] \tag{20.5}$$

Thus if we model the density of $\boldsymbol{z} \sim q(\boldsymbol{z}|\boldsymbol{x})$, which is a dequantized version of $\boldsymbol{x}$, we will get a lower bound on $p(\boldsymbol{x})$.

20.4.1.2 Likelihood can be hard to compute

Unfortunately, for many models, computing the likelihood can be computationally expensive, since it requires knowing the normalization constant of the probability model. One solution is to use variational inference (Chapter 10), which provides a way to efficiently compute lower (and sometimes

4. In some applications, we report **bits per dimension**, which is the log likelihood using log base 2, divided by the dimensionality of $\boldsymbol{x}$. To compute this metric, recall that $\log_2 L = \frac{\log_e L}{\log_e 2}$, and hence $\text{bpd} = \text{NLL} \log_e(2) \frac{1}{|\boldsymbol{x}|}$.

upper) bounds on the log likelihood. Another solution is to use annealed importance sampling (Section 11.5.4.1), which provides a way to estimate the log likelihood using Monte Carlo sampling. However, in the case of implicit generative models, such as GANs (Chapter 26), the likelihood is not even defined, so we need to find evaluation metrics that do not rely on likelihood.

20.4.1.3 Likelihood is not related to sample quality

A more subtle concern with likelihood is that it is often uncorrelated with the perceptual quality of the samples, at least for real-valued data, such as images and sound. In particular, a model can have great log-likelihood but create poor samples and vice versa.

To see why a model can have good likelihoods but create bad samples, consider the following argument from [TOB16]. Suppose q_0 is a density model for D-dimensional data $\boldsymbol{x}$ which performs arbitrarily well as judged by average log-likelihood, and suppose q_1 is a bad model, such as white noise. Now consider samples generated from the mixture model

$$q_2(\boldsymbol{x}) = 0.01 q_0(\boldsymbol{x}) + 0.99 q_1(\boldsymbol{x}) \tag{20.6}$$

Clearly 99% of the samples will be poor. However, the log-likelihood per pixel will hardly change between q_2 and q_0 if D is large, since

$$\log q_2(\boldsymbol{x}) = \log[0.01 q_0(\boldsymbol{x}) + 0.99 q_1(\boldsymbol{x})] \geq \log[0.01 q_0(\boldsymbol{x})] = \log q_0(\boldsymbol{x}) - 2 \tag{20.7}$$

For high-dimensional data, $|\log q_0(\boldsymbol{x})| \sim D \gg 100$, so $\log q_2(\boldsymbol{x}) \approx \log q_0(\boldsymbol{x})$, and hence mixing in the poor sampler does not significantly impact the log likelihood.

Now consider a case where the model has good samples but bad likelihoods. To achieve this, suppose q is a GMM centered on the training images:

$$q(\boldsymbol{x}) = \frac{1}{N} \sum_{n=1}^{N} \mathcal{N}(\boldsymbol{x} | \boldsymbol{x}_n, \epsilon^2 \mathbf{I}) \tag{20.8}$$

If ϵ is small enough that the Gaussian noise is imperceptible, then samples from this model will look good, since they correspond to the training set of real images. But this model will almost certainly have poor likelihood on the test set due to overfitting. (In this case we say the model has effectively just memorized the training set.)

20.4.2 Distances and divergences in feature space

Due to the challenges associated with comparing distributions in high dimensional spaces, and the desire to compare distributions in a semantically meaningful way, it is common to use domain-specific **perceptual distance metrics**, that measure how similar data vectors are to each other or to the training data. However, most metrics used to evaluate generative models do not directly compare raw data (e.g., pixels) but use a neural network to obtain features from the raw data and compare the feature distribution obtained from model samples with the feature distribution obtained from the dataset. The neural network used to obtain features can be trained solely for the purpose of evaluation, or can be pretrained; a common choice is to use a pretrained classifier (see e.g., [Sal+16; Heu+17b; Bin+18; Kyn+19; SSG18a]).

The **Inception score** [Sal+16] measures the average KL divergence between the marginal distribution of class labels obtained from the samples $p_{\boldsymbol{\theta}}(y) = \int p_{\text{disc}}(y|\boldsymbol{x}) p_{\boldsymbol{\theta}}(\boldsymbol{x}) d\boldsymbol{x}$ (where the integral is approximated by sampling images $\boldsymbol{x}$ from a fixed dataset) and the distribution $p(y|\boldsymbol{x})$ induced by samples from the model, $\boldsymbol{x} \sim p_{\boldsymbol{\theta}}(\boldsymbol{x})$. (The term comes from the "Inception" model [Sze+15b] that is often used to define $p_{\text{disc}}(y|\boldsymbol{x})$.) This leads to the following score:

$$\text{IS} = \exp\left[\mathbf{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{x})} D_{\mathbb{KL}}\left(p_{\text{disc}}(Y|\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(Y)\right)\right] \tag{20.9}$$

To understand this, let us rewrite the log score as follows:

$$\log(\text{IS}) = \mathbb{H}(p_{\boldsymbol{\theta}}(Y)) - \mathbb{E}_{p_{\boldsymbol{\theta}}(\boldsymbol{x})}\left[\mathbb{H}(p_{\text{disc}}(Y|\boldsymbol{x}))\right] \tag{20.10}$$

Thus we see that a high scoring model will be equally likely to generate samples from all classes, thus maximizing the entropy of $p_{\boldsymbol{\theta}}(Y)$, while also ensuring that each individual sample is easy to classify, thus minimizing the entropy of $p_{\text{disc}}(Y|\boldsymbol{x})$.

The Inception score solely relies on class labels, and thus does not measure overfitting or sample diversity outside the predefined dataset classes. For example, a model which generates one perfect example per class would get a perfect Inception score, despite not capturing the variety of examples inside a class, as shown in Figure 20.11a. To address this drawback, the **Fréchet Inception distance** or **FID** score [Heu+17b] measures the Fréchet distance between two Gaussian distributions on sets of features of a pre-trained classifier. One Gaussian is obtained by passing model samples through a pretrained classifier, and the other by passing dataset samples through the same classifier. If we assume that the mean and covariance obtained from model features are $\boldsymbol{\mu}_m$ and $\boldsymbol{\Sigma}_m$ and those from the data are $\boldsymbol{\mu}_d$ and $\boldsymbol{\Sigma}_d$, then the FID is

$$\text{FID} = \|\boldsymbol{\mu}_m - \boldsymbol{\mu}_d\|_2^2 + \text{tr}\Big(\boldsymbol{\Sigma}_d + \boldsymbol{\Sigma}_m - 2(\boldsymbol{\Sigma}_d \boldsymbol{\Sigma}_m)^{1/2}\Big) \tag{20.11}$$

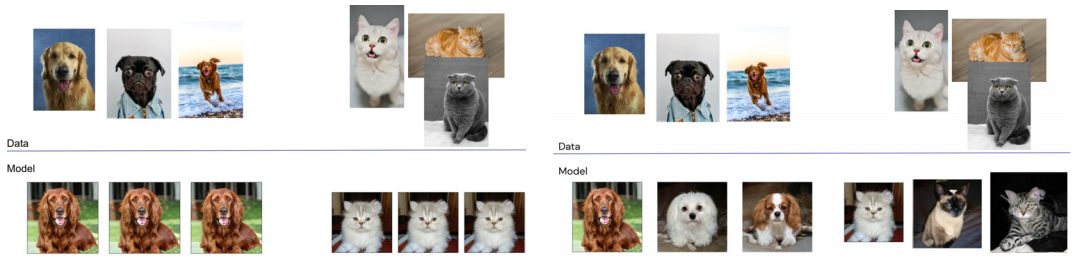
Since it uses features instead of class logits, the Fréchet distance captures more than modes captured by class labels, as shown in Figure 20.11b. Unlike the Inception score, a lower score is better since we want the two distributions to be as close as possible.

Unfortunately, the Fréchet distance has been shown to have a high bias, with results varying widely based on the number of samples used to compute the score. To mitigate this issue, the **kernel Inception distance** has been introduced [Bin+18], which measures the squared MMD (Section 2.7.3) between the features obtained from the data and features obtained from model samples.

20.4.3 Precision and recall metrics

Since the FID only measures the distance between the data and model distributions, it is difficult to use it as a diagnostic tool: a bad (high) FID can indicate that the model is not able to generate high quality data, or that it puts too much mass around the data distribution, or that the model only captures a subset of the data (e.g., in Figure 26.6). Trying to disentangle between these two failure modes has been the motivation to seek individual precision (sample quality) and recall (sample diversity) metrics in the context of generative models [LPO17; Kyn+19]. (The diversity question is especially important in the context of GANs, where mode collapse (Section 26.3.3) can be an issue.)

A common approach is to use nearest neighbors in the feature space of a pretrained classifier to



(a) (b)

Figure 20.11: (a) Model samples with good (high) inception score are visually realistic. (b) Model samples with good (low) FID score are visually realistic and diverse.

define precision and recall [Kyn+19]. To formalize this, let us define

$$f_k(\phi, \Phi) = \begin{cases} 1 & \text{if } \exists \phi' \in \Phi s.t. \|\phi - \phi'\|_2^2 \leq \|\phi' - \text{NN}_k(\phi', \Phi)\|_2^2 \\ 0 & \text{otherwise} \end{cases} \tag{20.12}$$

where Φ is a set of feature vectors and $\text{NN}_k(\phi', \Phi)$ is a function returning the k'th nearest neighbor of ϕ' in Φ. We now define precision and recall as follows:

$$\text{precision}(\Phi_{model}, \Phi_{data}) = \frac{1}{|\Phi_{model}|} \sum_{\phi \in \Phi_{model}} f_k(\phi, \Phi_{data}); \tag{20.13}$$

$$\text{recall}(\Phi_{model}, \Phi_{data}) = \frac{1}{|\Phi_{data}|} \sum_{\phi \in \Phi_{data}} f_k(\phi, \Phi_{model}); \tag{20.14}$$

Precision and recall are always between 0 and 1. Intuitively, the precision metric measures whether samples are as close to data as data is to other data examples, while recall measures whether data is as close to model samples as model samples are to other samples. The parameter k controls how lenient the metrics will be — the higher k, the higher both precision and recall will be. As in classification, precision and recall in generative models can be used to construct a trade-off curve between different models which allows practitioners to make an informed decision regarding which model they want to use.

20.4.4 Statistical tests

Statistical tests have long been used to determine whether two sets of samples have been generated from the same distribution; these types of statistical tests are called **two sample tests**. Let us define the null hypothesis as the statement that both set of samples are from the same distribution. We then compute a statistic from the data and compare it to a threshold, and based on this we decide whether to reject the null hypothesis. In the context of evaluating implicit generative models such as GANs, statistics based on classifiers [Saj+18] and the MMD [Liu+20b] have been used. For use in scenarios with high dimensional input spaces, which are ubiquitous in the era of deep learning, two sample tests have been adapted to use learned features instead of raw data.

Like all other evaluation metrics for generative models, statistical tests have their own advantages and disadvantages: while users can specify Type 1 error — the chance they allow that the null hypothesis is wrongly rejected — statistical tests tend to be computationally expensive and thus cannot be used to monitor progress in training; hence they are best used to compare fully trained models.

20.4.5 Challenges with using pretrained classifiers

While popular and convenient, evaluation metrics that rely on pretrained classifiers (such as IS, FID, nearest neighbors in feature space, and statistical tests in feature space) have significant drawbacks. One might not have a pretrained classifier available for the dataset at hand, so classifiers trained on other datasets are used. Given the well known challenges with neural network generalization (see Section 17.4), the features of a classifier trained on images from one dataset might not be reliable enough to provide a fine grained signal of quality for samples obtained from a model trained on a different dataset. If the generative model is trained on the same dataset as the pre-trained classifier but the model is not capturing the data distribution perfectly, we are presenting the pre-trained classifier with out-of-distribution data and relying on its features to obtain score to evaluate our models. Far from being purely theoretical concerns, these issues have been studied extensively and have been shown to affect evaluation in practice [RV19; BS18].

20.4.6 Using model samples to train classifiers

Instead of using pretrained classifiers to evaluate samples, one can train a classifier on samples from conditional generative models, and then see how good these classifiers are at classifying data. For example, does adding synthetic (sampled) data to the real data help? This is closer to a reliable evaluation of generative model samples, since ultimately, the performance of generative models is dependent on the downstream task they are trained for. If used for semisupervised learning, one should assess how much adding samples to a classifier dataset helps with test accuracy. If used for model based reinforcement learning, one should assess how much the generative model helps with agent performance. For examples of this approach, see e.g., [SSM18; SSA18; RV19; SS20b; Jor+22].

20.4.7 Assessing overfitting

Many of the metrics discussed so far capture the sample quality and diversity, but do not capture overfitting to the training data. To capture overfitting, often a visual inspection is performed: a set of samples is generated from the model and for each sample its closest K nearest neighbors in the feature space of a pretrained classifier are obtained from the dataset. While this approach requires manually assessing samples, it is a simple way to test whether a model is simply memorizing the data. We show an example in Figure 20.12: since the model sample in the top left is quite different than its neighbors from the dataset (remaining images), we can conclude the sample is not simply memorised from the dataset. Similarly, sample diversity can be measured by approximating the support of the learned distribution by looking for similar samples in a large sample pool — as in the pigeonhole principle — but it is expensive and often requires manual human assessment[AZ17].

For likelihood-based models — such as variational autoencoders (Chapter 21), autoregressive models (Chapter 22), and normalizing flows (Chapter 23) — we can assess memorization by seeing

Figure 20.12: Illustration of nearest neighbors in feature space: in the top left we have the query sample generated using BigGAN, and the rest of the images are its nearest neighbors from the dataset. The nearest neighbors search is done in the feature space of a pretrained classifier. From Figure 13 of [BDS18]. Used with kind permission of Andy Brock.

how much the log-likelihood of a model changes when a sample is included in the model's training set or not [BW21].

20.4.8 Human evaluation

One approach to evaluate generative models is to use human evaluation, by presenting samples from the model alongside samples from the data distribution, and ask human raters to compare the quality of the samples [Zho+19b]. Human evaluation is a suitable metric if the model is used to create art or other data for human display, or if reliable automated metrics are hard to obtain. However, human evaluation can be difficult to standardize, hard to automate, and can be expensive or cumbersome to set up.

21 Variational autoencoders

21.1 Introduction

In this chapter, we discuss generative models of the form

$$\boldsymbol{z} \sim p_{\boldsymbol{\theta}}(\boldsymbol{z}) \tag{21.1}$$

$$\boldsymbol{x}|\boldsymbol{z} \sim \text{Expfam}(\boldsymbol{x}|d_{\boldsymbol{\theta}}(\boldsymbol{z})) \tag{21.2}$$

where $p(\boldsymbol{z})$ is some kind of prior on the latent code $\boldsymbol{z}$, $d_{\boldsymbol{\theta}}(\boldsymbol{z})$ is a deep neural network, known as the **decoder**, and $\text{Expfam}(\boldsymbol{x}|\boldsymbol{\eta})$ is an exponential family distribution, such as a Gaussian or product of Bernoullis. This is called a **deep latent variable model** or **DLVM**. When the prior is Gaussian (as is often the case), this model is called a **deep latent Gaussian model** or **DLGM**.

Posterior inference (i.e., computing $p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})$) is computationally intractable, as is computing the marginal likelihood

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}) = \int p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) p_{\boldsymbol{\theta}}(\boldsymbol{z}) \, d\boldsymbol{z} \tag{21.3}$$

Hence we need to resort to approximate inference. For most of this chapter, we will use **amortized inference**, which we discussed in Section 10.1.5. This trains another model, $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$, called the **recognition network** or **inference network**, simultaneously with the generative model to do approximate posterior inference. This combination is called a **variational autoencoder** or **VAE** [KW14; RMW14b; KW19a], since it can be thought of as a probabilistic version of a deterministic autoencoder, discussed in Section 16.3.3.

In this chapter, we introduce the basic VAE, as well as some extensions. Note that the literature on VAE-like methods is vast[1], so we will only discuss a small subset of the ideas that have been explored.

21.2 VAE basics

In this section, we discuss the basics of variational autoencoders.

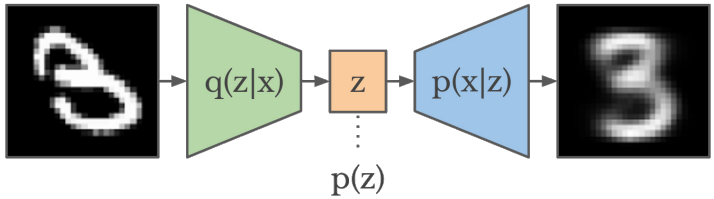


Figure 21.1: Schematic illustration of a VAE. From a figure in [Haf18]. Used with kind permission of Danijar Hafner.

21.2.1 Modeling assumptions

In the simplest setting, a VAE defines a generative model of the form

$$p_{\boldsymbol{\theta}}(\boldsymbol{z}, \boldsymbol{x}) = p_{\boldsymbol{\theta}}(\boldsymbol{z}) p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) \tag{21.4}$$

where $p_{\boldsymbol{\theta}}(\boldsymbol{z})$ is usually a Gaussian, and $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$ is usually a product of exponential family distributions (e.g., Gaussians or Bernoullis), with parameters computed by a neural network decoder, $d_{\boldsymbol{\theta}}(\boldsymbol{z})$. For example, for binary observations, we can use

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) = \prod_{d=1}^{D} \text{Ber}(x_d|\sigma(d_{\boldsymbol{\theta}}(\boldsymbol{z})) \tag{21.5}$$

In addition, a VAE fits a recognition model

$$q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) = q(\boldsymbol{z}|e_{\boldsymbol{\phi}}(\boldsymbol{x})) \approx p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}) \tag{21.6}$$

to perform approximate posterior inference. Here $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ is usually a Gaussian, with parameters computed by a neural network encoder $e_{\boldsymbol{\phi}}(\boldsymbol{x})$:

$$q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}, \text{diag}(\exp(\boldsymbol{\ell}))) \tag{21.7}$$

$$(\boldsymbol{\mu}, \boldsymbol{\ell}) = e_{\boldsymbol{\phi}}(\boldsymbol{x}) \tag{21.8}$$

where $\boldsymbol{\ell} = \log \boldsymbol{\sigma}$. The model can be thought of as encoding the input $\boldsymbol{x}$ into a stochastic latent bottleneck $\boldsymbol{z}$ and then decoding it to approximately reconstruct the input, as shown in Figure 21.1.

The idea of training an inference network to "invert" a generative network, rather than running an optimization algorithm to infer the latent code, is called amortized inference, and is discussed in Section 10.1.5. This idea was first proposed in the **Helmholtz machine** [Day+95]. However, that paper did not present a single unified objective function for inference and generation, but instead used the wake-sleep (Section 10.6) method for training. By contrast, the VAE optimizes a variational lower bound on the log-likelihood, which means that convergence to a locally optimal MLE of the parameters is guaranteed.

We can use other approaches to fitting the DLGM (see e.g., [Hof17; DF19]). However, learning an inference network to fit the DLGM is often faster and can have some regularization benefits (see e.g., [KP20]).[2]

1. For example, the website https://github.com/matthewvowels1/Awesome-VAEs lists over 900 papers.
2. Combining a generative model with an inference model in this way results in what has been called a "**monference**",

21.2.2 Model fitting

We can fit a VAE using amortized stochastic variational inference, as we discuss in Section 10.2.1.6. For example, suppose we use a VAE with a diagonal Bernoulli likelihood model, and a full covariance Gaussian as our variational posterior. Then we can use the methods discussed in Section 10.2.1.2 to derive the fitting algorith. See Algorithm 21.1 for the corresponding pseudocode.

Algorithm 21.1: Fitting a VAE with Bernoulli likelihood and full covariance Gaussian posterior. Based on Algorithm 2 of [KW19a].

Initialize $\boldsymbol{\theta}$, $\boldsymbol{\phi}$
repeat
  Sample $\boldsymbol{x} \sim p_{\mathcal{D}}$
  Sample $\boldsymbol{\epsilon} \sim q_0$
  $(\boldsymbol{\mu}, \log \boldsymbol{\sigma}, \mathbf{L}') = e_{\boldsymbol{\phi}}(\boldsymbol{x})$
  $\mathbf{M} = \text{np.triu}(\text{np.ones}(K), -1)$
  $\mathbf{L} = \mathbf{M} \odot \mathbf{L}' + \text{diag}(\boldsymbol{\sigma})$
  $\boldsymbol{z} = \mathbf{L}\boldsymbol{\epsilon} + \boldsymbol{\mu}$
  $\boldsymbol{p}_p = d_{\boldsymbol{\theta}}(\boldsymbol{z})$
  $\mathcal{L}_{\text{logqz}} = -\sum_{k=1}^{K} \left[\frac{1}{2}\epsilon_k^2 + \frac{1}{2}\log(2\pi) + \log \sigma_k\right]$ // from $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ in Equation (10.47)
  $\mathcal{L}_{\text{logpz}} = -\sum_{k=1}^{K} \left[\frac{1}{2}z_k^2 + \frac{1}{2}\log(2\pi)\right]$ // from $p_{\boldsymbol{\theta}}(\boldsymbol{z})$ in Equation (10.48)
  $\mathcal{L}_{\text{logpx}} = -\sum_{d=1}^{D} [x_d \log p_d + (1 - x_d)\log(1 - p_d)]$ // from $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$
  $\mathcal{L} = \mathcal{L}_{\text{logpx}} + \mathcal{L}_{\text{logpz}} - \mathcal{L}_{\text{logqz}}$
  Update $\boldsymbol{\theta} := \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}$
  Update $\boldsymbol{\phi} := \boldsymbol{\phi} - \eta \nabla_{\boldsymbol{\phi}} \mathcal{L}$
until converged

21.2.3 Comparison of VAEs and autoencoders

VAEs are very similar to deterministic autoencoders (AE). There are 2 main differences: in the AE, the objective is the log likelihood of the reconstruction without any KL term; and in addition, the encoding is deterministic, so the encoder network just needs to compute $\mathbb{E}[\boldsymbol{z}|\boldsymbol{x}]$ and not $\mathbb{V}[\boldsymbol{z}|\boldsymbol{x}]$. In view of these similarities, one can use the same codebase to implement both methods. However, it is natural to wonder what the benefits and potential drawbacks of the VAE are compared to the deterministic AE.

We shall answer this question by fitting both models to the CelebA dataset. Both models have the same convolutional structure with the following number of hidden channels per convolutional layer in the encoder: (32, 64, 128, 256, 512). The spatial size of each layer is as follows: (32, 16, 8, 4, 2). The final $2 \times 2 \times 512$ convolutional layer then gets reshaped and passed through a linear layer to generate the mean and (marginal) variance of the stochastic latent vector, which has size 256. The structure

i.e., model-inference hybrid. See the blog by Jacob Andreas, http://blog.jacobandreas.net/monference.html, for further discussion.

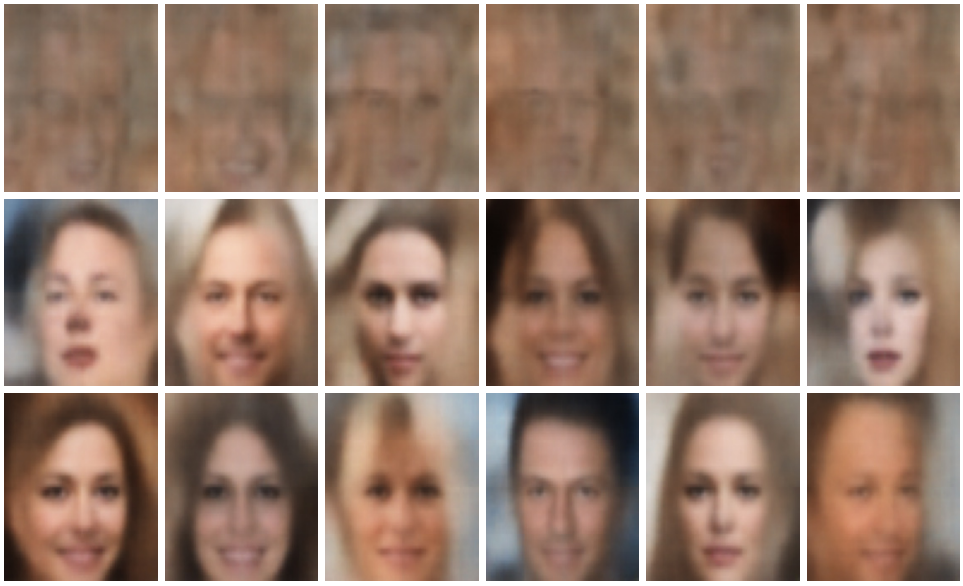


Figure 21.2: Illustration of unconditional image generation using (V)AEs trained on CelebA. Row 1: deterministic autoencoder. Row 2: β-VAE with $\beta = 0.5$. Row 3: VAE (with $\beta = 1$). Generated by celeba_vae_ae_comparison.ipynb.

of the decoder is the mirror image of the decoder. Each model is trained for 5 epochs with a batch size of 256, which takes about 20 minutes on a GPU.

The main advantage of a VAE over a deterministic autoencoder is that it defines a proper generative model, that can create sensible-looking novel images by decoding prior samples $\boldsymbol{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. By contrast, an autoencoder only knows how to decode latent codes derived from the training set, so does poorly when fed random inputs. This is illustrated in Figure 21.2.

We can also use both models to reconstruct a given input image. In Figure 21.3, we see that both AE and VAE can reconstruct the input images reasonably well, although the VAE reconstructions are somewhat blurry, for reasons we discuss in Section 21.3.1. We can reduce the amount of blurriness by scaling down the KL penalty term by a factor of β; this is known as the β-VAE, and is discussed in more detail in Section 21.3.1.

21.2.4 VAEs optimize in an augmented space

In this section, we derive several alternative expressions for the ELBO which shed light on how VAEs work.

First, let us define the joint generative distribution

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z}) = p_{\boldsymbol{\theta}}(\boldsymbol{z}) p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) \tag{21.9}$$

Figure 21.3: Illustration of image reconstruction using (V)AEs trained and applied to CelebA. Row 1: original images. Row 2: deterministic autoencoder. Row 3: β-VAE with $\beta = 0.5$. Row 4: VAE (with $\beta = 1$). Generated by celeba_vae_ae_comparison.ipynb.

from which we can derive the generative data marginal

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}) = \int_{\boldsymbol{z}} p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z}) d\boldsymbol{z} \tag{21.10}$$

and the generative posterior

$$p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}) = p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z}) / p_{\boldsymbol{\theta}}(\boldsymbol{x}) \tag{21.11}$$

Let us also define the joint *inference* distribution

$$q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{z}, \boldsymbol{x}) = p_{\mathcal{D}}(\boldsymbol{x}) q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \tag{21.12}$$

where

$$p_{\mathcal{D}}(\boldsymbol{x}) = \frac{1}{N} \sum_{n=1}^{N} \delta(\boldsymbol{x}_n - \boldsymbol{x}) \tag{21.13}$$

is the empirical distribution. From this we can derive the inference latent marginal, also called the **aggregated posterior**:

$$q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{z}) = \int_{\boldsymbol{x}} q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{x},\boldsymbol{z}) d\boldsymbol{x} \tag{21.14}$$

and the inference likelihood

$$q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{x}|\boldsymbol{z}) = q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{x},\boldsymbol{z}) / q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{z}) \tag{21.15}$$

See Figure 21.4 for a visual illustration.

Having defined our terms, we can now derive various alternative versions of the ELBO, following [ZSE19]. First note that the ELBO averaged over all the data is given by

$$\begin{aligned}
Ł(\boldsymbol{\theta},\boldsymbol{\phi}|\mathcal{D}) &= \mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x})}\left[\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right]\right] - \mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x})}\left[D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z})\right)\right] && (21.16)\\
&= \mathbb{E}_{q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{x},\boldsymbol{z})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) + \log p_{\boldsymbol{\theta}}(\boldsymbol{z}) - \log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})\right] && (21.17)\\
&= \mathbb{E}_{q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{x},\boldsymbol{z})}\left[\log \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z})}{q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{x},\boldsymbol{z})} + \log p_{\mathcal{D}}(\boldsymbol{x})\right] && (21.18)\\
&= -D_{\mathbb{KL}}\left(q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{x},\boldsymbol{z}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z})\right) + \mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x})}\left[\log p_{\mathcal{D}}(\boldsymbol{x})\right] && (21.19)
\end{aligned}$$

If we define $\stackrel{c}{=}$ to mean equal up to additive constants, we can rewrite the above as

$$\begin{aligned}
Ł(\boldsymbol{\theta},\boldsymbol{\phi}|\mathcal{D}) &\stackrel{c}{=} -D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{x},\boldsymbol{z}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z})\right) && (21.20)\\
&\stackrel{c}{=} -D_{\mathbb{KL}}\left(p_{\mathcal{D}}(\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{x})\right) - \mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x})}\left[D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})\right)\right] && (21.21)
\end{aligned}$$

Thus maximizing the ELBO requires minimizing the two KL terms. The first KL term is minimized by MLE, and the second KL term is minimized by fitting the true posterior. Thus if the posterior family is limited, there may be a conflict between these objectives.

Finally, we note that the ELBO can also be written as

$$Ł(\boldsymbol{\theta},\boldsymbol{\phi}|\mathcal{D}) \stackrel{c}{=} -D_{\mathbb{KL}}\left(q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{z}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z})\right) - \mathbb{E}_{q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{z})}\left[D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{x}|\boldsymbol{z}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right)\right] \tag{21.22}$$

We see from Equation (21.22) that VAEs are trying to minimize the difference between the inference marginal and generative prior, $D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z})\right)$, while simultaneously minimizing reconstruction error, $D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{x}|\boldsymbol{z}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right)$ Since $\boldsymbol{x}$ is typically of much higher dimensionality than $\boldsymbol{z}$, the latter term usually dominates. Consequently, if there is a conflict between these two objectives (e.g., due to limited modeling power), the VAE will favor reconstruction accuracy over posterior inference. Thus the learned posterior may not be a very good approximation to the true posterior (see [ZSE19] for further discussion).

21.3 VAE generalizations

In this section, we discuss some variants of the basic VAE model.

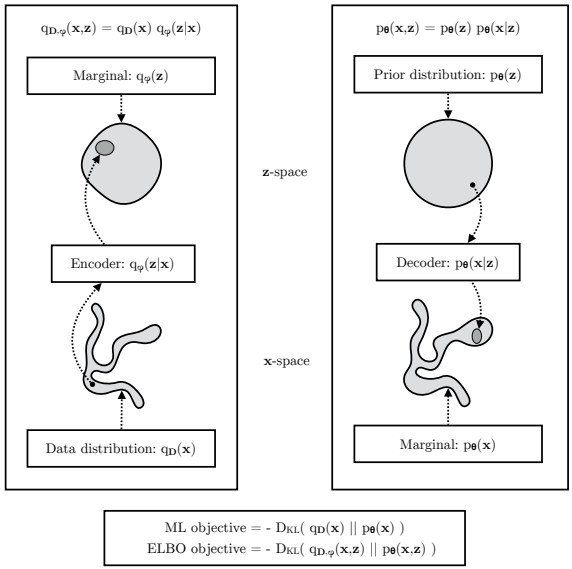


Figure 21.4: The maximum likelihood (ML) objective can be viewed as the minimization of $D_{\mathbb{KL}}\left(p_{\mathcal{D}}(\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{x})\right)$. (Note: in the figure, $p_{\mathcal{D}}(\boldsymbol{x})$ is denoted by $q_D(\boldsymbol{x})$.) The ELBO objective is minimization of $D_{\mathbb{KL}}\left(q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{x},\boldsymbol{z}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z})\right)$, which upper bounds $D_{\mathbb{KL}}\left(q_{\mathcal{D}}(\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{x})\right)$. From Figure 2.4 of [KW19a]. Used with kind permission of Durk Kingma.

21.3.1 β-VAE

It is often the case that VAEs generate somewhat blurry images, as illustrated in Figure 21.3, Figure 21.2 and Figure 20.9. This is not the case for models that optimize the exact likelihood, such as pixelCNNs (Section 22.3.2) and flow models (Chapter 23). To see why VAEs are different, consider the common case where the decoder is a Gaussian with fixed variance, so

$$\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) = -\frac{1}{2\sigma^2}||\boldsymbol{x} - d_{\boldsymbol{\theta}}(\boldsymbol{z})||_2^2 + \text{const} \tag{21.23}$$

Let $e_{\boldsymbol{\phi}}(\boldsymbol{x}) = \mathbb{E}\left[q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})\right]$ be the encoding of $\boldsymbol{x}$, and $\mathcal{X}(\boldsymbol{z}) = \{\boldsymbol{x} : e_{\boldsymbol{\phi}}(\boldsymbol{x}) = \boldsymbol{z}\}$ be the set of inputs that get mapped to $\boldsymbol{z}$. For a fixed inference network, the optimal setting of the generator parameters, when using squared reconstruction loss, is to ensure $d_{\boldsymbol{\theta}}(\boldsymbol{z}) = \mathbb{E}\left[\boldsymbol{x} : \boldsymbol{x} \in \mathcal{X}(\boldsymbol{z})\right]$. Thus the decoder should predict the average of all inputs $\boldsymbol{x}$ that map to that $\boldsymbol{z}$, resulting in blurry images.

We can solve this problem by increasing the expressive power of the posterior approximation (avoiding the merging of distinct inputs into the same latent code), or of the generator (by adding back information that is missing from the latent code), or both. However, an even simpler solution is to reduce the penalty on the KL term, making the model closer to a deterministic autoencoder:

$$\mathcal{L}_\beta(\boldsymbol{\theta}, \boldsymbol{\phi}|\boldsymbol{x}) = \underbrace{-\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right]}_{\mathcal{L}_E} + \beta \underbrace{D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z})\right)}_{\mathcal{L}_R} \tag{21.24}$$

where $\mathcal{L}_E$ is the reconstruction error (negative log likelihood), and $\mathcal{L}_R$ is the KL regularizer. This is

called the β-**VAE** objective [Hig+17a]. If we set $\beta = 1$, we recover the objective used in standard VAEs; if we set $\beta = 0$, we recover the objective used in standard autoencoders.

By varying β from 0 to infinity, we can reach different points on the **rate distortion curve**, as discussed in Section 5.4.2. These points make different tradeoffs between reconstruction error (distortion) and how much information is stored in the latents about the input (rate of the corresponding code). By using $\beta < 1$, we store more bits about each input, and hence can reconstruct images in a less blurry way. If we use $\beta > 1$, we get a more compressed representation.

21.3.1.1 Disentangled representations

One advantage of using $\beta > 1$ is that it encourages the learning of a latent representation that is "**disentangled**". Intuitively this means that each latent dimension represents a different **factor of variation** in the input. This is often formalized in terms of the total correlation (Section 5.3.5.1), which is defined as follows:

$$\mathrm{TC}(\boldsymbol{z}) = \sum_k \mathbb{H}(z_k) - \mathbb{H}(\boldsymbol{z}) = D_{\mathbb{KL}}\left(p(\boldsymbol{z}) \parallel \prod_k p_k(z_k)\right) \tag{21.25}$$

This is zero iff the components of $\boldsymbol{z}$ are all mutually independent, and hence disentangled. In [AS18], they prove that using $\beta > 1$ will decrease the TC.

Unfortunately, in [Loc+18] they prove that nonlinear latent variable models are unidentifiable, and therefore for any disentangled representation, there is an equivalent fully entangled representation with exactly the same likelihood. Thus it is not possible to recover the correct latent representation without choosing the appropriate inductive bias, via the encoder, decoder, prior, dataset, or learning algorithm, i.e., merely adjusting β is not sufficient. See Section 32.4.1 for more discussion.

21.3.1.2 Connection with information bottleneck

In this section, we show that the β-VAE is an unsupervised version of the information bottleneck (IB) objective from Section 5.6. If the input is $\boldsymbol{x}$, the hidden bottleneck is $\boldsymbol{z}$, and the target outputs are $\tilde{\boldsymbol{x}}$, then the unsupervised IB objective becomes

$$\mathcal{L}_{\mathrm{UIB}} = \beta\, \mathbb{I}(\boldsymbol{z}; \boldsymbol{x}) - \mathbb{I}(\boldsymbol{z}; \tilde{\boldsymbol{x}}) \tag{21.26}$$

$$= \beta \mathbb{E}_{p(\boldsymbol{x},\boldsymbol{z})}\left[\log \frac{p(\boldsymbol{x},\boldsymbol{z})}{p(\boldsymbol{x})p(\boldsymbol{z})}\right] - \mathbb{E}_{p(\boldsymbol{z},\tilde{\boldsymbol{x}})}\left[\log \frac{p(\boldsymbol{z},\tilde{\boldsymbol{x}})}{p(\boldsymbol{z})p(\tilde{\boldsymbol{x}})}\right] \tag{21.27}$$

where

$$p(\boldsymbol{x},\boldsymbol{z}) = p_{\mathcal{D}}(\boldsymbol{x})p(\boldsymbol{z}|\boldsymbol{x}) \tag{21.28}$$

$$p(\boldsymbol{z},\tilde{\boldsymbol{x}}) = \int p_{\mathcal{D}}(\boldsymbol{x})p(\boldsymbol{z}|\boldsymbol{x})p(\tilde{\boldsymbol{x}}|\boldsymbol{z})d\boldsymbol{x} \tag{21.29}$$

Intuitively, the objective in Equation (21.26) means we should pick a representation $\boldsymbol{z}$ that can predict $\tilde{\boldsymbol{x}}$ reliably, while not memorizing too much information about the input $\boldsymbol{x}$. The tradeoff parameter is controlled by β.

From Equation (5.181), we have the following variational upper bound on this unsupervised objective:

$$\mathcal{L}_{\text{UVIB}} = -\mathbb{E}_{q_{\mathcal{D},\boldsymbol{\phi}}(\boldsymbol{z},\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right] + \beta \mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x})}\left[D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z})\right)\right] \tag{21.30}$$

which matches Equation (21.24) when averaged over $\boldsymbol{x}$.

21.3.2 InfoVAE

In Section 21.2.4, we discussed some drawbacks of the standard ELBO objective for training VAEs, namely the tendency to ignore the latent code when the decoder is powerful (Section 21.4), and the tendency to learn a poor posterior approximation due to the mismatch between the KL terms in data space and latent space (Section 21.2.4). We can fix these problems to some degree by using a generalized objective of the following form:

$$Ł(\boldsymbol{\theta}, \boldsymbol{\phi}|\boldsymbol{x}) = -\lambda D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z})\right) - \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z})}\left[D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{x}|\boldsymbol{z}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right)\right] + \alpha\, \mathbb{I}_q(\boldsymbol{x}; \boldsymbol{z}) \tag{21.31}$$

where $\alpha \geq 0$ controls how much we weight the mutual information $\mathbb{I}_q(\boldsymbol{x}; \boldsymbol{z})$ between $\boldsymbol{x}$ and $\boldsymbol{z}$, and $\lambda \geq 0$ controls the tradeoff between $\boldsymbol{z}$-space KL and $\boldsymbol{x}$-space KL. This is called the **InfoVAE** objective [ZSE19]. If we set $\alpha = 0$ and $\lambda = 1$, we recover the standard ELBO, as shown in Equation (21.22).

Unfortunately, the objective in Equation (21.31) cannot be computed as written, because of the intractable MI term:

$$\mathbb{I}_q(\boldsymbol{x}; \boldsymbol{z}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{x},\boldsymbol{z})}\left[\log \frac{q_{\boldsymbol{\phi}}(\boldsymbol{x}, \boldsymbol{z})}{q_{\boldsymbol{\phi}}(\boldsymbol{x}) q_{\boldsymbol{\phi}}(\boldsymbol{z})}\right] = -\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{x},\boldsymbol{z})}\left[\log \frac{q_{\boldsymbol{\phi}}(\boldsymbol{z})}{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\right] \tag{21.32}$$

However, using the fact that $q_{\boldsymbol{\phi}}(\boldsymbol{x}|\boldsymbol{z}) = p_{\mathcal{D}}(\boldsymbol{x}) q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) / q_{\boldsymbol{\phi}}(\boldsymbol{z})$, we can rewrite the objective as follows:

$$Ł = \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{x},\boldsymbol{z})}\left[-\lambda \log \frac{q_{\boldsymbol{\phi}}(\boldsymbol{z})}{p_{\boldsymbol{\theta}}(\boldsymbol{z})} - \log \frac{q_{\boldsymbol{\phi}}(\boldsymbol{x}|\boldsymbol{z})}{p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})} - \alpha \log \frac{q_{\boldsymbol{\phi}}(\boldsymbol{z})}{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\right] \tag{21.33}$$

$$= \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{x},\boldsymbol{z})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) - \log \frac{q_{\boldsymbol{\phi}}(\boldsymbol{z})^{\lambda+\alpha-1} p_{\mathcal{D}}(\boldsymbol{x})}{p_{\boldsymbol{\theta}}(\boldsymbol{z})^{\lambda} q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})^{\alpha-1}}\right] \tag{21.34}$$

$$= \mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x})}\left[\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right]\right] - (1-\alpha)\mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x})}\left[D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z})\right)\right]$$
$$- (\alpha + \lambda - 1) D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z})\right) - \mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x})}\left[\log p_{\mathcal{D}}(\boldsymbol{x})\right] \tag{21.35}$$

where the last term is a constant we can ignore. The first two terms can be optimized using the reparameterization trick. Unfortunately, the last term requires computing $q_{\boldsymbol{\phi}}(\boldsymbol{z}) = \int_{\boldsymbol{x}} q_{\boldsymbol{\phi}}(\boldsymbol{x}, \boldsymbol{z}) d\boldsymbol{x}$, which is intractable. Fortunately, we can easily sample from this distribution, by sampling $\boldsymbol{x} \sim p_{\mathcal{D}}(\boldsymbol{x})$ and $\boldsymbol{z} \sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$. Thus $q_{\boldsymbol{\phi}}(\boldsymbol{z})$ is an **implicit probability model**, similar to a GAN (see Chapter 26).

As long as we use a strict divergence, meaning $D(q, p) = 0$ iff $q = p$, then one can show that this does not affect the optimality of the procedure. In particular, proposition 2 of [ZSE19] tells us the following:

Theorem 1. *Let $\mathcal{X}$ and $\mathcal{Z}$ be continuous spaces, and $\alpha < 1$ (to bound the MI) and $\lambda > 0$. For any fixed value of $\mathbb{I}_q(\boldsymbol{x}; \boldsymbol{z})$, the approximate InfoVAE loss, with any strict divergence $D(q_{\boldsymbol{\phi}}(\boldsymbol{z}), p_{\boldsymbol{\theta}}(\boldsymbol{z}))$, is globally optimized if $p_{\boldsymbol{\theta}}(\boldsymbol{x}) = p_{\mathcal{D}}(\boldsymbol{x})$ and $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) = p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})$.*

21.3.2.1 Connection with MMD VAE

If we set $\alpha = 1$, the InfoVAE objective simplifies to

$$Ł \stackrel{c}{=} \mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x})}\left[\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right]\right] - \lambda D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z})\right) \tag{21.36}$$

The **MMD VAE**[3] replaces the KL divergence in the above term with the (squared) maximum mean discrepancy or **MMD** divergence defined in Section 2.7.3. (This is valid based on the above theorem.) The advantage of this approach over standard InfoVAE is that the resulting objective is tractable. In particular, if we set $\lambda = 1$ and swap the sign we get

$$\mathcal{L} = \mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x})}\left[\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[-\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right]\right] + \text{MMD}(q_{\boldsymbol{\phi}}(\boldsymbol{z}), p_{\boldsymbol{\theta}}(\boldsymbol{z})) \tag{21.37}$$

As we discuss in Section 2.7.3, we can compute the MMD as follows:

$$\text{MMD}(p, q) = \mathbb{E}_{p(\boldsymbol{z}),p(\boldsymbol{z}')}\left[\mathcal{K}(\boldsymbol{z}, \boldsymbol{z}')\right] + \mathbb{E}_{q(\boldsymbol{z}),q(\boldsymbol{z}')}\left[\mathcal{K}(\boldsymbol{z}, \boldsymbol{z}')\right] - 2\mathbb{E}_{p(\boldsymbol{z}),q(\boldsymbol{z}')}\left[\mathcal{K}(\boldsymbol{z}, \boldsymbol{z}')\right] \tag{21.38}$$

where $\mathcal{K}()$ is some kernel function, such as the RBF kernel, $\mathcal{K}(\boldsymbol{z}, \boldsymbol{z}') = \exp(-\frac{1}{2\sigma^2}||\boldsymbol{z} - \boldsymbol{z}'||_2^2)$. Intuitively the MMD measures the similarity (in latent space) between samples from the prior and samples from the aggregated posterior.

In practice, we can implement the MMD objective by using the posterior predicted mean $\boldsymbol{z}_n = e_{\boldsymbol{\phi}}(\boldsymbol{x}_n)$ for all B samples in the current minibatch, and comparing this to B random samples from the $\mathcal{N}(\mathbf{0}, \mathbf{I})$ prior.

If we use a Gaussian decoder with fixed variance, the negative log likelihood is just a squared error term:

$$-\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) = ||\boldsymbol{x} - d_{\boldsymbol{\theta}}(\boldsymbol{z})||_2^2 \tag{21.39}$$

Thus the entire model is deterministic, and just predicts the means in latent space and visible space.

21.3.2.2 Connection with β-VAEs

If we set $\alpha = 0$ and $\lambda = 1$, we get back the original ELBO. If $\lambda > 0$ is freely chosen, but we use $\alpha = 1 - \lambda$, we get the β-VAE.

21.3.2.3 Connection with adversarial autoencoders

If we set $\alpha = 1$ and $\lambda = 1$, and D is chosen to be the Jensen-Shannon divergence (which can be minimized by training a binary discriminator, as explained in Section 26.2.2), then we get a model known as an **adversarial autoencoder** [Mak+15a].

21.3.3 Multimodal VAEs

It is possible to extend VAEs to create joint distributions over different kinds of variables, such as images and text. This is sometimes called a **multimodal VAE** or **MVAE**. Let us assume there are

3. Proposed in https://ermongroup.github.io/blog/a-tutorial-on-mmd-variational-autoencoders/.

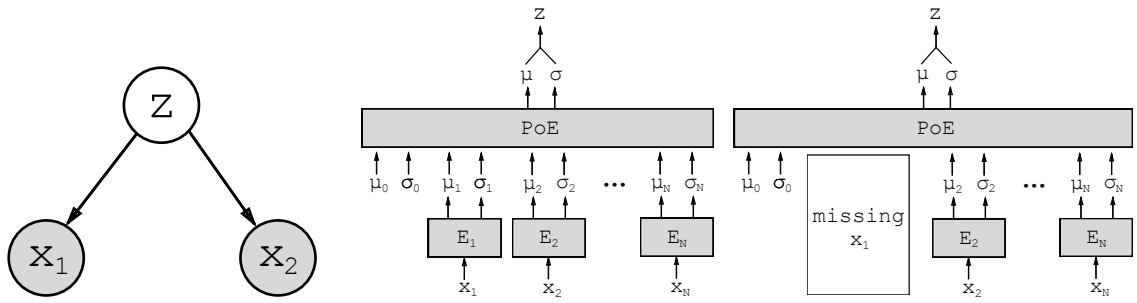


Figure 21.5: Illustration of multi-modal VAE. (a) The generative model with $N = 2$ modalities. (b) The product of experts (PoE) inference network is derived from N individual Gaussian experts E_i. μ_0 and σ_0 are parameters of the prior. (c) If a modality is missing, we omit its contribution to the posterior. From Figure 1 of [WG18]. Used with kind permission of Mike Wu.

M modalities. We assume they are conditionally independent given the latent code, and hence the generative model has the form

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M, \boldsymbol{z}) = p(\boldsymbol{z}) \prod_{m=1}^{M} p_{\boldsymbol{\theta}}(\boldsymbol{x}_m|\boldsymbol{z}) \tag{21.40}$$

where we treat $p(\boldsymbol{z})$ as a fixed prior. See Figure 21.5(a) for an illustration.

The standard ELBO is given by

$$Ł(\boldsymbol{\theta}, \boldsymbol{\phi}|\mathbf{X}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\mathbf{X})}\left[\sum_m \log p_{\boldsymbol{\theta}}(\boldsymbol{x}_m|\boldsymbol{z})\right] - D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\mathbf{X}) \parallel p(\boldsymbol{z})\right) \tag{21.41}$$

where $\mathbf{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M)$ is the observed data. However, the different likelihood terms $p(\boldsymbol{x}_m|\boldsymbol{z})$ may have different dynamic ranges (e.g., Gaussian pdf for pixels, and categorical pmf for text), so we introduce weight terms $\lambda_m \geq 0$ for each likelihood. In addition, let $\beta \geq 0$ control the amount of KL regularization. This gives us a weighted version of the ELBO, as follows:

$$Ł(\boldsymbol{\theta}, \boldsymbol{\phi}|\mathbf{X}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\mathbf{X})}\left[\sum_m \lambda_m \log p_{\boldsymbol{\theta}}(\boldsymbol{x}_m|\boldsymbol{z})\right] - \beta D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\mathbf{X}) \parallel p(\boldsymbol{z})\right) \tag{21.42}$$

Often we don't have a lot of paired (aligned) data from all M modalities. For example, we may have a lot of images (modality 1), and a lot of text (modality 2), but very few (image, text) pairs. So it is useful to generalize the loss so it fits the marginal distributions of subsets of the features. Let $O_m = 1$ if modality m is observed (i.e., $\boldsymbol{x}_m$ is known), and let $O_m = 0$ if it is missing or unobserved. Let $\mathbf{X} = \{\boldsymbol{x}_m : O_m = 1\}$ be the visible features. We now use the following objective:

$$Ł(\boldsymbol{\theta}, \boldsymbol{\phi}|\mathbf{X}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\mathbf{X})}\left[\sum_{m:O_m=1} \lambda_m \log p_{\boldsymbol{\theta}}(\boldsymbol{x}_m|\boldsymbol{z})\right] - \beta D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\mathbf{X}) \parallel p(\boldsymbol{z})\right) \tag{21.43}$$

The key problem is how to compute the posterior $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\mathbf{X})$ given different subsets of features. In general this can be hard, since the inference network is a discriminative model that assumes all inputs are available. For example, if it is trained on (image, text) pairs, $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}_1, \boldsymbol{x}_2)$, how can we compute the posterior just given an image, $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}_1)$, or just given text, $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}_2)$? (This issue arises in general with VAE when we have missing inputs.)

Fortunately, based on our conditional independence assumption between the modalities, we can compute the optimal form for $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\mathbf{X})$ given set of inputs by computing the exact posterior under the model, which is given by

$$p(\boldsymbol{z}|\mathbf{X}) = \frac{p(\boldsymbol{z})p(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M|\boldsymbol{z})}{p(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M)} = \frac{p(\boldsymbol{z})}{p(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M)} \prod_{m=1}^{M} p(\boldsymbol{x}_m|\boldsymbol{z}) \tag{21.44}$$

$$= \frac{p(\boldsymbol{z})}{p(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M)} \prod_{m=1}^{M} \frac{p(\boldsymbol{z}|\boldsymbol{x}_m)p(\boldsymbol{x}_m)}{p(\boldsymbol{z})} \tag{21.45}$$

$$\propto p(\boldsymbol{z}) \prod_{m=1}^{M} \frac{p(\boldsymbol{z}|\boldsymbol{x}_m)}{p(\boldsymbol{z})} \approx p(\boldsymbol{z}) \prod_{m=1}^{M} \tilde{q}(\boldsymbol{z}|\boldsymbol{x}_m) \tag{21.46}$$

This can be viewed as a product of experts (Section 24.1.1), where each $\tilde{q}(\boldsymbol{z}|\boldsymbol{x}_m)$ is an "expert" for the m'th modality, and $p(\boldsymbol{z})$ is the prior. We can compute the above posterior for any subset of modalities for which we have data by modifying the product over m. If we use Gaussian distributions for the prior $p(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}_0, \boldsymbol{\Lambda}_0^{-1})$ and marginal posterior ratio $\tilde{q}(\boldsymbol{z}|\boldsymbol{x}_m) = \mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}_m, \boldsymbol{\Lambda}_m^{-1})$, then we can compute the product of Gaussians using the result from Equation (2.154):

$$\prod_{m=0}^{M} \mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}_m, \boldsymbol{\Lambda}_m^{-1}) \propto \mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}, \boldsymbol{\Sigma}), \quad \boldsymbol{\Sigma} = (\sum_m \boldsymbol{\Lambda}_m)^{-1}, \quad \boldsymbol{\mu} = \boldsymbol{\Sigma}(\sum_m \boldsymbol{\Lambda}_m \boldsymbol{\mu}_m) \tag{21.47}$$

Thus the overall posterior precision is the sum of individual expert posterior precisions, and the overall posterior mean is the precision weighted average of the individual expert posterior means. See Figure 21.5(b) for an illustration. For a linear Gaussian (factor analysis) model, we can ensure $q(\boldsymbol{z}|\boldsymbol{x}_m) = p(\boldsymbol{z}|\boldsymbol{x}_m)$, in which case the above solution is the exact posterior [WN18], but in general it will be an approximation.

We need to train the individual expert recognition models $q(\boldsymbol{z}|\boldsymbol{x}_m)$ as well as the joint model $q(\boldsymbol{z}|\mathbf{X})$, so the model knows what to do with fully observed as well as partially observed inputs at test time. In [Ved+18], they propose a somewhat complex "triple ELBO" objective. In [WG18], they propose the simpler approach of optimizing the ELBO for the fully observed feature vector, all the marginals, and a set of $\mathcal{J}$ randomly chosen joint modalities:

$$Ł(\boldsymbol{\theta}, \boldsymbol{\phi}|\mathbf{X}) = Ł(\boldsymbol{\theta}, \boldsymbol{\phi}|()\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M) + \sum_{m=1}^{M} Ł(\boldsymbol{\theta}, \boldsymbol{\phi}|\boldsymbol{x}_m) + \sum_{j \in \mathcal{J}} Ł(\boldsymbol{\theta}, \boldsymbol{\phi}|\mathbf{X}_j) \tag{21.48}$$

This generalizes nicely to the semi-supervised setting, in which we only have a few aligned ("labeled") examples from the joint, but have many unaligned ("unlabeled") examples from the individual marginals. See Figure 21.5(c) for an illustration.

Note that the above scheme can only handle the case of a fixed number of missingness patterns; we can generalize to allow for arbitrary missingness as discussed in [CNW20]. (See also Section 3.11 for a more general discussion of missing data.)

21.3.4 Semisupervised VAEs

In this section, we discuss how to extend VAEs to the **semi-supervised learning** setting in which we have both labeled data, $\mathcal{D}_L = \{(\boldsymbol{x}_n, y_n)\}$, and unlabeled data, $\mathcal{D}_U = \{(\boldsymbol{x}_n)\}$. We focus on the **M2** model, proposed in [Kin+14a].

The generative model has the following form:

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}, y) = p_{\boldsymbol{\theta}}(y)p_{\boldsymbol{\theta}}(\boldsymbol{x}|y) = p_{\boldsymbol{\theta}}(y) \int p_{\boldsymbol{\theta}}(\boldsymbol{x}|y, \boldsymbol{z})p_{\boldsymbol{\theta}}(\boldsymbol{z})d\boldsymbol{z} \tag{21.49}$$

where $\boldsymbol{z}$ is a latent variable, $p_{\boldsymbol{\theta}}(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{z}|\mathbf{0}, \mathbf{I})$ is the latent prior, $p_{\boldsymbol{\theta}}(y) = \text{Cat}(y|\boldsymbol{\pi})$ the label prior, and $p_{\boldsymbol{\theta}}(\boldsymbol{x}|y, \boldsymbol{z}) = p(\boldsymbol{x}|f_{\boldsymbol{\theta}}(y, \boldsymbol{z}))$ is the likelihood, such as a Gaussian, with parameters computed by f (a deep neural network). The main innovation of this approach is to assume that data is generated according to both a latent class variable y as well as the continuous latent variable $\boldsymbol{z}$. The class variable y is observed for labeled data and unobserved for unlabled data.

To compute the likelihood for the *labeled data*, $p_{\boldsymbol{\theta}}(\boldsymbol{x}, y)$, we need to marginalize over $\boldsymbol{z}$, which we can do by using an inference network of the form

$$q_{\boldsymbol{\phi}}(\boldsymbol{z}|y, \boldsymbol{x}) = \mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}_{\boldsymbol{\phi}}(y, \boldsymbol{x}), \text{diag}(\boldsymbol{\sigma}_{\boldsymbol{\phi}}(y, \boldsymbol{x})) \tag{21.50}$$

We then use the following variational lower bound

$$\log p_{\boldsymbol{\theta}}(\boldsymbol{x}, y) \geq \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x},y)} \left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|y, \boldsymbol{z}) + \log p_{\boldsymbol{\theta}}(y) + \log p_{\boldsymbol{\theta}}(\boldsymbol{z}) - \log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}, y)\right] = -\mathcal{L}(\boldsymbol{x}, y) \tag{21.51}$$

as is standard for VAEs (see Section 21.2). The only difference is that we observe two kinds of data: $\boldsymbol{x}$ and y.

To compute the likelihood for the *unlabeled* data, $p_{\boldsymbol{\theta}}(\boldsymbol{x})$, we need to marginalize over $\boldsymbol{z}$ *and* y, which we can do by using an inference network of the form

$$q_{\boldsymbol{\phi}}(\boldsymbol{z}, y|\boldsymbol{x}) = q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})q_{\boldsymbol{\phi}}(y|\boldsymbol{x}) \tag{21.52}$$

$$q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}_{\boldsymbol{\phi}}(\boldsymbol{x}), \text{diag}(\boldsymbol{\sigma}_{\boldsymbol{\phi}}(\boldsymbol{x})) \tag{21.53}$$

$$q_{\boldsymbol{\phi}}(y|\boldsymbol{x}) = \text{Cat}(y|\boldsymbol{\pi}_{\boldsymbol{\phi}}(\boldsymbol{x})) \tag{21.54}$$

Note that $q_{\boldsymbol{\phi}}(y|\boldsymbol{x})$ acts like a discriminative classifier, that imputes the missing labels. We then use the following variational lower bound:

$$\log p_{\boldsymbol{\theta}}(\boldsymbol{x}) \geq \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z},y|\boldsymbol{x})} \left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|y, \boldsymbol{z}) + \log p_{\boldsymbol{\theta}}(y) + \log p_{\boldsymbol{\theta}}(\boldsymbol{z}) - \log q_{\boldsymbol{\phi}}(\boldsymbol{z}, y|\boldsymbol{x})\right] \tag{21.55}$$

$$= -\sum_y q_{\boldsymbol{\phi}}(y|\boldsymbol{x})\mathcal{L}(\boldsymbol{x}, y) + \mathbb{H}\left(q_{\boldsymbol{\phi}}(y|\boldsymbol{x})\right) = -\mathcal{U}(\boldsymbol{x}) \tag{21.56}$$

Note that the discriminative classifier $q_{\boldsymbol{\phi}}(y|\boldsymbol{x})$ is only used to compute the log-likelihood of the unlabeled data, which is undesirable. We can therefore add an extra classification loss on the supervised data, to get the following overall objective function:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}_L} \left[\mathcal{L}(\boldsymbol{x}, y)\right] + \mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}_U} \left[\mathcal{U}(\boldsymbol{x})\right] + \alpha\mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}_L} \left[-\log q_{\boldsymbol{\phi}}(y|\boldsymbol{x})\right] \tag{21.57}$$

where $\mathcal{D}_L$ is the labeled data, $\mathcal{D}_U$ is the unlabeled data, and α is a hyperparameter that controls the relative weight of generative and discriminative learning.

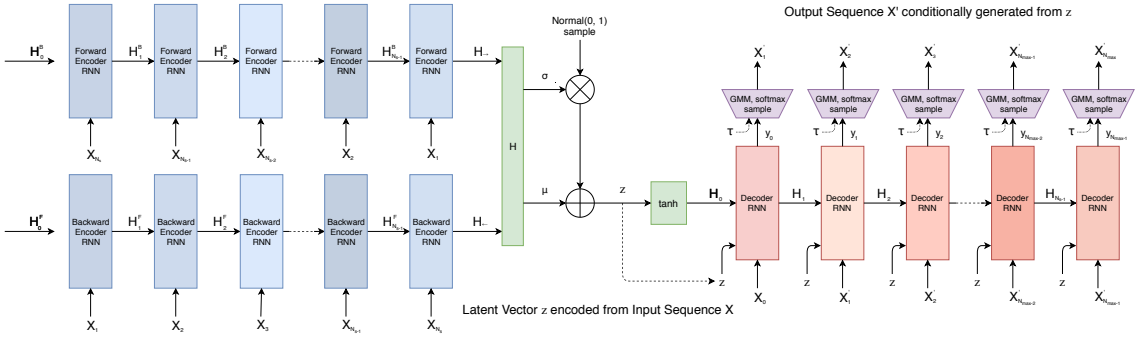


Figure 21.6: Illustration of a VAE with a bidirectional RNN encoder and a unidirectional RNN decoder. The output generator can use a GMM and/or softmax distribution. From Figure 2 of [HE18]. Used with kind permission of David Ha.

21.3.5 VAEs with sequential encoders/decoders

In this section, we discuss VAEs for sequential data, such as text and biosequences, in which the data $\boldsymbol{x}$ is a variable-length sequence, but we have a fixed-sized latent variable $\boldsymbol{z} \in \mathbb{R}^K$. (We consider the more general case in which $\boldsymbol{z}$ is a variable-length sequence of latents — known as **sequential VAE** or **dynamic VAE** — in Section 29.13.) All we have to do is modify the decoder $p(\boldsymbol{x}|\boldsymbol{z})$ and encoder $q(\boldsymbol{z}|\boldsymbol{x})$ to work with sequences.

21.3.5.1 Models

If we use an RNN for the encoder and decoder of a VAE, we get a model which is called a **VAE-RNN**, as proposed in [Bow+16a]. In more detail, the generative model is $p(\boldsymbol{z}, \boldsymbol{x}_{1:T}) = p(\boldsymbol{z})\text{RNN}(\boldsymbol{x}_{1:T}|\boldsymbol{z})$, where $\boldsymbol{z}$ can be injected as the initial state of the RNN, or as an input to every time step. The inference model is $q(\boldsymbol{z}|\boldsymbol{x}_{1:T}) = \mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}(\boldsymbol{h}), \boldsymbol{\Sigma}(\boldsymbol{h}))$, where $\boldsymbol{h} = [\boldsymbol{h}_T^{\rightarrow}, \boldsymbol{h}_1^{\leftarrow}]$ is the output of a bidirectional RNN applied to $\boldsymbol{x}_{1:T}$. See Figure 21.6 for an illustration.

More recently, people have tried to combine transformers with VAEs. For example, in the **Optimus** model of [Li+20], they use a BERT model for the encoder. In more detail, the encoder $q(\boldsymbol{z}|\boldsymbol{x})$ is derived from the embedding vector associated with a dummy token corresponding to the "class label" which is appended to the input sequence $\boldsymbol{x}$. The decoder is a standard autoregressive model (similar to GPT), with one additional input, namely the latent vector $\boldsymbol{z}$. They consider two ways of injecting the latent vector. The simplest approach is to add $\boldsymbol{z}$ to the embedding layer of every token in the decoding step, by defining $\boldsymbol{h}'_i = \boldsymbol{h}_i + \mathbf{W}\boldsymbol{z}$, where $\boldsymbol{h}_i \in \mathbb{R}^H$ is the original embedding for the i'th token, and $\mathbf{W} \in \mathbb{R}^{H \times K}$ is a decoding matrix, where K is the size of the latent vector. However, they get better results in their experiments by letting all the layers of the decoder attend to the latent code $\boldsymbol{z}$. An easy way to do this is to define the memory vector $\boldsymbol{h}_m = \mathbf{W}\boldsymbol{z}$, where $\mathbf{W} \in \mathbb{R}^{LH \times K}$, where L is the number of layers in the decoder, and then to append $\boldsymbol{h}_m \in \mathbb{R}^{L \times H}$ to all the other embeddings at each layer.

An alternative approach, known as **transformer VAE**, was proposed in [Gre20]. This model uses a **funnel transformer** [Dai+20b] as the encoder, and the **T5** [Raf+20a] conditional transformer for

```
he was silent for a long moment .
he was silent for a moment .
it was quiet for a moment .
it was dark and cold .
there was a pause .
it was my turn .
```

(a)

```
i went to the store to buy some groceries .
i store to buy some groceries .
i were to buy any groceries .
horses are to buy any groceries .
horses are to buy any animal .
horses the favorite any animal .
horses the favorite favorite animal .
horses are my favorite animal .
```

(b)

Figure 21.7: (a) Samples from the latent space of a VAE text model, as we interpolate between two sentences (on first and last line). Note that the intermediate sentences are grammatical, and semantically related to their neighbors. From Table 8 of [Bow+16b]. (b) Same as (a), but now using a deterministic autoencoder (with the same RNN encoder and decoder). From Table 1 of [Bow+16b]. Used with kind permission of Sam Bowman.

the decoder. In addition, it uses an MMD VAE (Section 21.3.2.1) to avoid posterior collapse.

21.3.5.2 Applications

In this section, we discuss some applications of VAEs to sequence data.

Text

In [Bow+16b], they apply the VAE-RNN model to natural language sentences. (See also [MB16; SSB17] for related work.) Although this does not improve performance in terms of the standard perplexity measures (predicting the next word given the previous words), it does provide a way to infer a semantic representation of the sentence. This can then be used for latent space interpolation, as discussed in Section 20.3.5. The results of doing this with the VAE-RNN are illustrated in Figure 21.7a. (Similar results are shown in [Li+20], using a VAE-transformer.) By contrast, if we use a standard deterministic autoencoder, with the same RNN encoder and decoder networks, we learn a much less meaningful space, as illustrated in Figure 21.7b. The reason is that the deterministic autoencoder has "holes" in its latent space, which get decoded to nonsensical outputs.

However, because RNNs (and transformers) are powerful decoders, we need to address the problem of posterior collapse, which we discuss in Section 21.4. One common way to avoid this problem is to use KL annealing, but a more effective method is to use the InfoVAE method of Section 21.3.2, which includes adversarial autoencoders (used in [She+20] with an RNN decoder) and MMD autoencoders (used in [Gre20] with a transformer decoder).

Sketches

In [HE18], they apply the VAE-RNN model to generate sketches (line drawings) of various animals and hand-written characters. They call their model **sketch-rnn**. The training data records the sequence of (x, y) pen positions, as well as whether the pen was touching the paper or not. The emission model used a GMM for the real-valued location offsets, and a categorical softmax distribution for the discrete state.

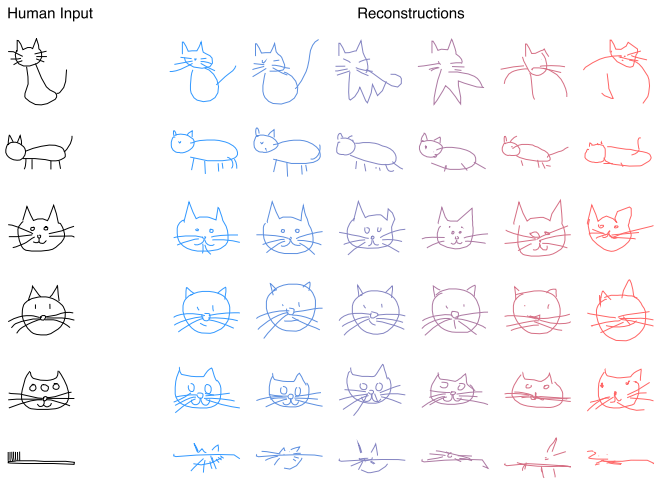


Figure 21.8: Conditional generation of cats from sketch-RNN model. We increase the temperature parameter from left to right. From Figure 5 of [HE18]. Used with kind permission of David Ha.

Figure 21.8 shows some samples from various class-conditional models. We vary the temperature parameter τ of the emission model to control the stochasticity of the generator. (More precisely, we multiply the GMM variances by τ, and divide the discrete probabilities by τ before renormalizing.) When the temperature is low, the model tries to reconstruct the input as closely as possible. However, when the input is untypical of the training set (e.g., a cat with three eyes, or a toothbrush), the reconstruction is "regularized" towards a canonical cat with two eyes, while still keeping some features of the input.

Molecular design

In [GB+18], they use VAE-RNNs to model molecular graph structure, represented as a string using the SMILES representation.[4] It is also possible to learn a mapping from the latent space to some scalar quantity of interest, such as the solubility or drug efficacy of a molecule. We can then perform gradient-based optimization in the continuous latent space to try to generate new graphs which maximize this quantity. See Figure 21.9 for a sketch of this approach.

The main problem is to ensure that points in latent space decode to valid strings/molecules. There are various solutions to this, including using a **grammar VAE**, where the RNN decoder is replaced by a stochastic context free grammar. See [KPHL17] for details.

21.4 Avoiding posterior collapse

If the decoder $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$ is sufficiently powerful (e.g., a pixel CNN, or an RNN for text), then the VAE does not need to use the latent code $\boldsymbol{z}$ for anything. This is called **posterior collapse** or **variational**

4. See https://en.wikipedia.org/wiki/Simplified_molecular-input_line-entry_system.

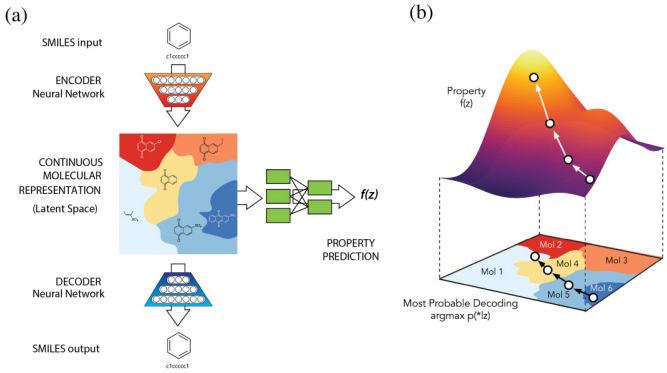


Figure 21.9: Application of VAE-RNN to molecule design. (a) The VAE-RNN model is trained on a sequence representation of molecules known as SMILES. We can fit an MLP to map from the latent space to properties of the molecule, such as its "fitness" $f(\boldsymbol{z})$. (b) We can perform gradient ascent in $f(\boldsymbol{z})$ space, and then decode the result to a new molecule with high fitness. From Figure 1 of [GB+18]. Used with kind permission of Rafael Gomez-Bombarelli.

overpruning (see e.g., [Che+17b; Ale+18; Hus17a; Phu+18; TT17; Yeu+17; Luc+19; DWW19; WBC21]). To see why this happens, consider Equation (21.21). If there exists a parameter setting for the generator $\boldsymbol{\theta}^*$ such that $p_{\boldsymbol{\theta}^*}(\boldsymbol{x}|\boldsymbol{z}) = p_{\mathcal{D}}(\boldsymbol{x})$ for every $\boldsymbol{z}$, then we can make $D_{\mathbb{KL}}\left(p_{\mathcal{D}}(\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{x})\right) = 0$. Since the generator is independent of the latent code, we have $p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}) = p_{\boldsymbol{\theta}}(\boldsymbol{z})$. The prior $p_{\boldsymbol{\theta}}(\boldsymbol{z})$ is usually a simple distribution, such as a Gaussian, so we can find a setting of the inference parameters so that $q_{\boldsymbol{\phi}^*}(\boldsymbol{z}|\boldsymbol{x}) = p_{\boldsymbol{\theta}}(\boldsymbol{z})$, which ensures $D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})\right) = 0$. Thus we have succesfully maximized the ELBO, but we have not learned any useful latent representation of the data, which is one of the goals of latent variable modeling.[5] We discuss some solutions to posterior collapse below.

21.4.1 KL annealing

A common approach to solving this problem, proposed in [Bow+16a], is to use **KL annealing**, in which the KL penalty term in the ELBO is scaled by β, which is increased from 0.0 (corresponding to an autoencoder) to 1.0 (which corresponds to standard MLE training). (Note that, by contrast, the β-VAE model in Section 21.3.1 uses $\beta > 1$.)

KL annealing can work well, but requires tuning the schedule for β. A standard practice [Fu+19] is to use **cyclical annealing**, which repeats the process of increasing β multiple times. This ensures the progressive learning of more meaningful latent codes, by leveraging good representations learned in a previous cycle as a way to warmstart the optimization.

5. Note that [Luc+19; DWW20] show that posterior collapse can also happen in linear VAE models, where the ELBO corresponds to the exact marginal likelihood, so the problem is not only due to powerful (nonlinear) decoders, but is also related to spurious local maxima in the objective.

21.4.2 Lower bounding the rate

An alternative approach is to stick with the original unmodified ELBO objective, but to prevent the rate (i.e., the $D_{\mathbb{KL}}\left(q \parallel p\right)$ term) from collapsing to 0, by limiting the flexibility of q. For example, [XD18; Dav+18] use a von Mises-Fisher (Section 2.2.5.3) prior and posterior, instead of a Gaussian, and they constrain the posterior to have a fixed concentration, $q(\boldsymbol{z}|\boldsymbol{x}) = \text{vMF}(\boldsymbol{z}|\boldsymbol{\mu}(\boldsymbol{x}), \kappa)$. Here the parameter κ controls the rate of the code. The **δ-VAE** method [Oor+19] uses a Gaussian autoregressive prior and a diagonal Gaussian posterior. We can ensure the rate is at least δ by adjusting the regression parameter of the AR prior.

21.4.3 Free bits

In this section, we discuss the method of **free bits** [Kin+16], which is another way of lower bounding the rate. To explain this, consider a fully factorized posterior in which the KL penalty has the form

$$\mathcal{L}_R = \sum_i D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(z_i|\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(z_i)\right) \tag{21.58}$$

where z_i is the i'th dimension of $\boldsymbol{z}$. We can replace this with a hinge loss, that will give up driving down the KL for dimensions that are already beneath a target compression rate λ:

$$\mathcal{L}'_R = \sum_i \max(\lambda, D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(z_i|\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(z_i)\right)) \tag{21.59}$$

Thus the bits where the KL is sufficiently small "are free", since the model does not have to "pay" to encode them according to the prior.

21.4.4 Adding skip connections

One reason for latent variable collapse is that the latent variables $\boldsymbol{z}$ are not sufficiently "connected to" the observed data $\boldsymbol{x}$. One simple solution is to modify the architecture of the generative model by adding **skip connections**, similar to a residual network (Section 16.2.4), as shown in Figure 21.10. This is called a **skip-VAE** [Die+19a].

21.4.5 Improved variational inference

The posterior collapse problem is caused in part by the poor approximation to the posterior. In [He+19], they proposed to keep the model and VAE objective unchanged, but to more aggressively update the inference network before each step of generative model fitting. This enables the inference network to capture the current true posterior more faithfully, which will encourage the generator to use the latent codes when it is useful to do so.

However, this only addresses the part of posterior collapse that is due to the amortization gap [CLD18], rather than the more fundamental problem of variational pruning, in which the KL term penalizes the model if its posterior deviates too far from the prior, which is often too simple to match the aggregated posterior.

Another way to ameliorate variational pruning is to use lower bounds that are tighter than the vanilla ELBO (Section 10.5.1), or more accurate posterior approximations (Section 10.4), or more accurate (hierarchical) generative models (Section 21.5).

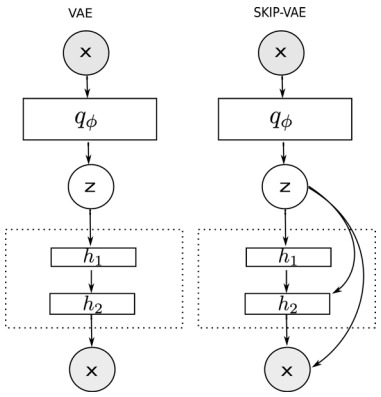


Figure 21.10: (a) VAE. (b) Skip-VAE. From Figure 1 of [Die+19a]. Used with kind permission of Adji Dieng.

21.4.6 Alternative objectives

An alternative to the above methods is to replace the ELBO objective with other objectives, such as the InfoVAE objective discussed in Section 21.3.2, which includes adversarial autoencoders and MMD autoencoders as special cases. The InfoVAE objective includes a term to explicitly enforce non-zero mutual information between $\boldsymbol{x}$ and $\boldsymbol{z}$, which effectively solves the problem of posterior collapse.

21.5 VAEs with hierarchical structure

We define a **hierarchical VAE** or HVAE, with L stochastic layers, to be the following generative model:[6]

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z}_{1:L}) = p_{\boldsymbol{\theta}}(\boldsymbol{z}_L) \left[\prod_{l=L-1}^{1} p_{\boldsymbol{\theta}}(\boldsymbol{z}_l | \boldsymbol{z}_{l+1}) \right] p_{\boldsymbol{\theta}}(\boldsymbol{x} | \boldsymbol{z}_1) \tag{21.60}$$

We can improve on the above model by making it non-Markovian, i.e., letting each $\boldsymbol{z}_l$ depend on all the higher level stochastic variables, $\boldsymbol{z}_{l+1:L}$, not just the preceeding level, i.e.,

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{z}) = p_{\boldsymbol{\theta}}(\boldsymbol{z}_L) \left[\prod_{l=L-1}^{1} p_{\boldsymbol{\theta}}(\boldsymbol{z}_l | \boldsymbol{z}_{l+1:L}) \right] p_{\boldsymbol{\theta}}(\boldsymbol{x} | \boldsymbol{z}_{1:L}) \tag{21.61}$$

Note that the likelihood is now $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}_{1:L})$ instead of just $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}_1)$. This is analogous to adding skip connections from all preceeding variables to all their children. It is easy to implement this by using a deterministic "backbone" of residual connections, that accumulates all stochastic decisions, and propagates them down the chain, as illustrated in Figure 21.11(left). We discuss how to perform inference and learning in such models below.

6. There is a split in the literature about whether to label the top level as $\boldsymbol{z}_L$ or $\boldsymbol{z}_1$. We adopt the former convention, since we view lower numbered layers, such as $\boldsymbol{z}_1$, as being "closer to the data", and higher numbered layers, such as $\boldsymbol{z}_L$, as being "more abstract".

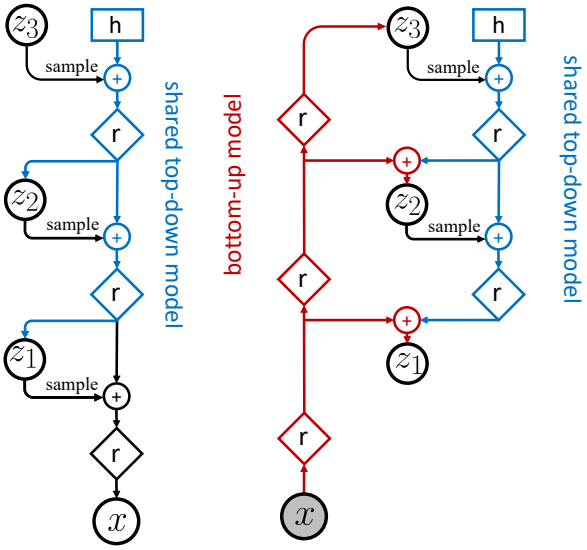


Figure 21.11: Hierarchical VAEs with 3 stochastic layers. Left: generative model. Right: inference network. Diamond is a residual network, $\oplus$ is feature combination (e.g., concatenation), and h is a trainable parameter. We first do bottom-up inference, by propagating $\boldsymbol{x}$ up to $\boldsymbol{z}_3$ to compute $\boldsymbol{z}_3^s \sim q_{\boldsymbol{\phi}}(\boldsymbol{z}_3|\boldsymbol{x})$, and then we perform top-down inference by computing $\boldsymbol{z}_2^s \sim q_{\boldsymbol{\phi}}(\boldsymbol{z}_2|\boldsymbol{x}, \boldsymbol{z}_3^s)$ and then $\boldsymbol{z}_1^s \sim q_{\boldsymbol{\phi}}(\boldsymbol{z}_1|\boldsymbol{x}, \boldsymbol{z}_{2:3}^s)$. From Figure 2 of [VK20a]. Used with kind permission of Arash Vahdat.

21.5.1 Bottom-up vs top-down inference

To perform inference in a hierarchical VAE, we could use a **bottom-up inference model** of the form

$$q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) = q_{\boldsymbol{\phi}}(\boldsymbol{z}_1|\boldsymbol{x}) \prod_{l=2}^{L} q_{\boldsymbol{\phi}}(\boldsymbol{z}_l|\boldsymbol{x}, \boldsymbol{z}_{1:l-1}) \tag{21.62}$$

However, a better approach is to use a **top-down inference model** of the form

$$q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) = q_{\boldsymbol{\phi}}(\boldsymbol{z}_L|\boldsymbol{x}) \prod_{l=L-1}^{1} q_{\boldsymbol{\phi}}(\boldsymbol{z}_l|\boldsymbol{x}, \boldsymbol{z}_{l+1:L}) \tag{21.63}$$

Inference for $\boldsymbol{z}_l$ combines bottom-up information from $\boldsymbol{x}$ with top-down information from higher layers, $\boldsymbol{z}_{>l} = \boldsymbol{z}_{l+1:L}$. See Figure 21.11(right) for an illustration.[7]

7. Note that it is also possible to have a stochastic bottom-up encoder and a stochastic top-down encoder, as discussed in the **BIVA** paper [Maa+19]. (BIVA stands for "bidirectional-inference variational autoencoder".)

With the above model, the ELBO can be written as follows (using the chain rule for KL):

$$Ł(\boldsymbol{\theta}, \boldsymbol{\phi}|\boldsymbol{x}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right] - D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}_L|\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z}_L)\right) \tag{21.64}$$

$$- \sum_{l=L-1}^{1} \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}_{>l}|\boldsymbol{x})}\left[D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}_l|\boldsymbol{x}, \boldsymbol{z}_{>l}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z}_l|\boldsymbol{z}_{>l})\right)\right] \tag{21.65}$$

where

$$q_{\boldsymbol{\phi}}(\boldsymbol{z}_{>l}|\boldsymbol{x}) = \prod_{i=l+1}^{L} q_{\boldsymbol{\phi}}(\boldsymbol{z}_i|\boldsymbol{x}, \boldsymbol{z}_{>i}) \tag{21.66}$$

is the approximate posterior above layer l (i.e., the parents of $\boldsymbol{z}_l$).

The reason the top-down inference model is better is that it more closely approximates the true posterior of a given layer, which is given by

$$p_{\boldsymbol{\theta}}(\boldsymbol{z}_l|\boldsymbol{x}, \boldsymbol{z}_{l+1:L}) \propto p_{\boldsymbol{\theta}}(\boldsymbol{z}_l|\boldsymbol{z}_{l+1:L}) p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}_l, \boldsymbol{z}_{l+1:L}) \tag{21.67}$$

Thus the posterior combines the top-down prior term $p_{\boldsymbol{\theta}}(\boldsymbol{z}_l|\boldsymbol{z}_{l+1:L})$ with the bottom-up likelihood term $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}_l, \boldsymbol{z}_{l+1:L})$. We can approximate this posterior by defining

$$q_{\boldsymbol{\phi}}(\boldsymbol{z}_l|\boldsymbol{x}, \boldsymbol{z}_{l+1:L}) \propto p_{\boldsymbol{\theta}}(\boldsymbol{z}_l|\boldsymbol{z}_{l+1:L}) \tilde{q}_{\boldsymbol{\phi}}(\boldsymbol{z}_l|\boldsymbol{x}, \boldsymbol{z}_{l+1:L}) \tag{21.68}$$

where $\tilde{q}_{\boldsymbol{\phi}}(\boldsymbol{z}_l|\boldsymbol{x}, \boldsymbol{z}_{l+1:L})$ is a learned Gaussian approximation to the bottom-up likelihood. If both prior and likelihood are Gaussian, we can compute this product in closed form, as proposed in the **ladder network** paper [Sn+16; Søn+16].[8] A more flexible approach is to let $q_{\boldsymbol{\phi}}(\boldsymbol{z}_l|\boldsymbol{x}, \boldsymbol{z}_{l+1:L})$ be learned, but to force it to share some of its parameters with the learned prior $p_{\boldsymbol{\theta}}(\boldsymbol{z}_l|\boldsymbol{z}_{l+1:L})$, as proposed in [Kin+16]. This reduces the number of parameters in the model, and ensures that the posterior and prior remain somewhat close.

21.5.2 Example: very deep VAE

There have been many papers exploring different kinds of HVAE models (see e.g., [Kin+16; Sn+16; Chi21a; VK20a; Maa+19]), and we do not have space to discuss them all. Here we focus on the "**very deep VAE**" or **VD-VAE** model of [Chi21a], since it is simple but yields state of the art results (at the time of writing).

The architecture is a simple convolutional VAE with bidirectional inference, as shown in Figure 21.12. For each layer, the prior and posterior are diagonal Gaussians. The author found that nearest-neighbor upsampling (in the decoder) worked much better than transposed convolution, and avoided posterior collapse. This enabled training with the vanilla VAE objective, without needing any of the tricks discussed in Section 21.5.4.

The low-resolution latents (at the top of the hierarchy) capture a lot of the global structure of each image; the remaining high-resolution latents are just used to fill in details, that make the image look more realistic, and improve the likelihood. This suggests the model could be useful for lossy

8. The term "ladder network" arises from the horizontal "rungs" in Figure 21.11(right). Note that a similar idea was independently proposed in [Sal16].

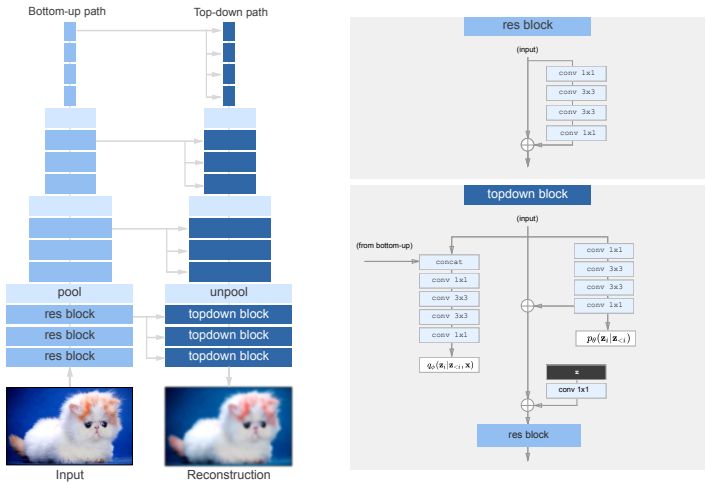


Figure 21.12: The top-down encoder used by the hierarchical VAE in [Chi21a]. Each convolution is preceded by the GELU nonlinearity. The model uses average pooling and nearest-neighbor upsampling for the pool and unpool layers. The posterior $q_{\boldsymbol{\phi}}$ and prior $p_{\boldsymbol{\theta}}$ are diagonal Gaussians. From Figure 3 of [Chi21a]. Used with kind permission of Rewon Child.

Figure 21.13: Samples from a VDVAE model (trained on FFHQ dataset) from different levels of the hierarchy. From Figure 1 of [Chi21a]. Used with kind permission of Rewon Child.

compression, since a lot of the low-level details can be drawn from the prior (i.e., "hallucinated"), rather than having to be sent by the encoder.

We can also use the model for unconditional sampling at multiple resolutions. This is illustrated in Figure 21.13, using a model with 78 stochastic layers trained on the FFHQ-256 dataset.[9].

21.5.3 Connection with autoregressive models

Until recently, most hierarchical VAEs only had a small number of stochastic layers. Consequently the images they generated have not looked as good, or had as high likelihoods, as images produced by other models, such as the autoregressive PixelCNN model (see Section 22.3.2). However, by endowing VAEs with many more stochastic layers, it is possible to outperform AR models in terms of

9. This is a 256^2 version of the Flickr-Faces High Quality dataset from https://github.com/NVlabs/ffhq-dataset, which has 80k images at 1024^2 resoution.

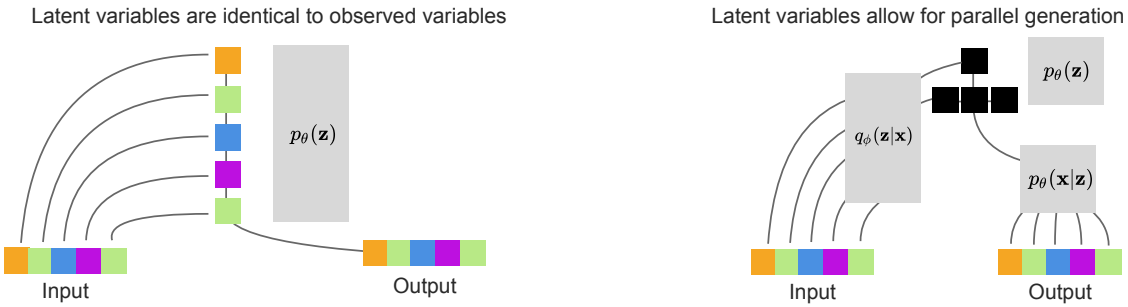


Figure 21.14: Left: a hierarchical VAE which emulates an autoregressive model using an identify encoder, autoregressive prior, and identity decoder. Right: a hierarchical VAE with a 2 layer hierarchical latent code. The bottom hidden nodes (black) are conditionally independent given the top layer. From Figure 2 of [Chi21a]. Used with kind permission of Rewon Child.

likelihood and sample quality, while using fewer parameters and much less computing power [Chi21a; VK20a; Maa+19].

To see why this is possible, note that we can represent any AR model as a degenerate VAE, as shown in Figure 21.14(left). The idea is simple: the encoder copies the input into latent space by setting $\boldsymbol{z}_{1:D} = \boldsymbol{x}_{1:D}$ (so $q_{\boldsymbol{\phi}}(z_i = x_i|\boldsymbol{z}_{>i}, \boldsymbol{x}) = 1$), then the model learns an autoregressive prior $p_{\boldsymbol{\theta}}(\boldsymbol{z}_{1:D}) = \prod_d p(z_d|\boldsymbol{z}_{1:d-1})$, and finally the likelihood function just copies the latent vector to output space, so $p_{\boldsymbol{\theta}}(x_i = z_i|\boldsymbol{z}) = 1$. Since the encoder computes the exact (albeit degenerate) posterior, we have $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) = p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})$, so the ELBO is tight and reduces to the log likelihood,

$$\log p_{\boldsymbol{\theta}}(\boldsymbol{x}) = \log p_{\boldsymbol{\theta}}(\boldsymbol{z}) = \sum_d \log p_{\boldsymbol{\theta}}(x_d|\boldsymbol{x}_{<d}) \tag{21.69}$$

Thus we can emulate any AR model with a VAE providing it has at least D stochastic layers, where D is the dimensionality of the observed data.

In practice, data usually lives in a lower-dimensional manifold (see e.g., [DW19]), which can allow for a much more compact latent code. For example, Figure 21.14(right) shows a hierarchical code in which the latent factors at the lower level are conditionally independent given the higher level, and hence can be generated in parallel. Such a tree-like structure can enable sample generation in $O(\log D)$ time, whereas an autoregressive model always takes $O(D)$ time. (Recall that for an image D is the number of pixels, so it grows quadratically with image resolution. For example, even a tiny 32×32 image has $D = 3072$.)

In addition to speed, hierarchical models also require many fewer parameters than "flat" models. The typical architecture used for generating images is a **multi-scale** approach: the model starts from a small, spatially arranged set of latent variables, and at each subsequent layer, the spatial resolution is increased (usually by a factor of 2). This allows the high level to capture global, long-range correlations (e.g., the symmetry of a face, or overall skin tone), while letting lower levels capture fine-grained details.

21.5.4 Variational pruning

A common problem with hierarchical VAEs is that the higher level latent layers are often ignored, so the model does not learn interesting high level semantics. This is caused by **variational pruning**. This problem is analogous to the issue of latent variable collapse, which we discussed in Section 21.4.

A common heuristic to mitigate this problem is to use KL balancing coefficients [Che+17b], to ensure that an equal amount of information is encoded in each layer. That is, we use the following penalty:

$$\sum_{l=1}^{L} \gamma_l \mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}_{>l}|\boldsymbol{x})} \left[D_{\mathbb{KL}} \left(q_{\boldsymbol{\phi}}(\boldsymbol{z}_l|\boldsymbol{x}, \boldsymbol{z}_{>l}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z}_l|\boldsymbol{z}_{>l}) \right) \right] \tag{21.70}$$

The balancing term γ_l is set to a small value when the KL penalty is small (on the current minibatch), to encourage use of that layer, and is set to a large value when the KL term is large. (This is only done during the "warm up period".) Concretely, [VK20a] proposes to set the coefficients γ_l to be proportional to the size of the layer, s_l, and the average KL loss:

$$\gamma_l \propto s_l \mathbb{E}_{\boldsymbol{x} \sim \mathcal{B}} \left[\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}_{>l}|\boldsymbol{x})} \left[D_{\mathbb{KL}} \left(q_{\boldsymbol{\phi}}(\boldsymbol{z}_l|\boldsymbol{x}, \boldsymbol{z}_{>l}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z}_l|\boldsymbol{z}_{>l}) \right) \right] \right] \tag{21.71}$$

where $\mathcal{B}$ is the current minibatch.

21.5.5 Other optimization difficulties

A common problem when training (hierarchical) VAEs is that the loss can become unstable. The main reason for this is that the KL term is unbounded (can become infinitely large). In [Chi21a], they tackle the problem in two ways. First, ensure the initial random weights of the final convolutional layer in each residual bottleneck block get scaled by $1/\sqrt{L}$. Second, skip an update step if the norm of the gradient of the loss exceeds some threshold.

In the **Nouveau VAE** method of [VK20a], they use some more complicated measures to ensure stability. First, they use batch normalization, but with various tweaks. Second, they use spectral regularization for the encoder. Specifically they add the penalty $\beta \sum_i \lambda_i$, where λ_i is the largest singular value of the i'th convolutional layer (estimated using a single power iteration step), and $\beta \geq 0$ is a tuning parameter. Third, they use inverse autoregressive flows (Section 23.2.4.3) in each layer, instead of a diagonal Gaussian approximation. Fourth, they represent the posterior using a residual representation. In particular, let us assume the prior for the i'th variable in layer l is

$$p_{\boldsymbol{\theta}}(z_l^i|\boldsymbol{z}_{>l}) = \mathcal{N}(z_l^i|\mu_i(\boldsymbol{z}_{>l}), \sigma_i(\boldsymbol{z}_{>l})) \tag{21.72}$$

They propose the following posterior approximation:

$$q_{\boldsymbol{\phi}}(z_l^i|\boldsymbol{x}, \boldsymbol{z}_{>l}) = \mathcal{N}\left(z_l^i|\mu_i(\boldsymbol{z}_{>l}) + \Delta\mu_i(\boldsymbol{z}_{>l}, \boldsymbol{x}),\ \sigma_i(\boldsymbol{z}_{>l}) \cdot \Delta\sigma_i(\boldsymbol{z}_{>l}, \boldsymbol{x})\right) \tag{21.73}$$

where the Δ terms are the relative changes computed by the encoder. The corresponding KL penalty reduces to the following (dropping the l subscript for brevity):

$$D_{\mathbb{KL}}\left(q_{\boldsymbol{\phi}}(z^i|\boldsymbol{x}, \boldsymbol{z}_{>l}) \parallel p_{\boldsymbol{\theta}}(z^i|\boldsymbol{z}_{>l})\right) = \frac{1}{2}\left(\frac{\Delta\mu_i^2}{\sigma_i^2} + \Delta\sigma_i^2 - \log \Delta\sigma_i^2 - 1\right) \tag{21.74}$$

So as long as σ_i is bounded from below, the KL term can be easily controlled just by adjusting the encoder parameters.

Figure 21.15: Autoencoder for MNIST using 256 binary latents. Top row: input images. Middle row: reconstruction. Bottom row: latent code, reshaped to a 16 × 16 image. Generated by quantized_autoencoder_mnist.ipynb.

21.6 Vector quantization VAE

In this section, we describe **VQ-VAE**, which stands for "vector quantized VAE" [OVK17; ROV19]. This is like a standard VAE except it uses a set of discrete latent variables.

21.6.1 Autoencoder with binary code

The simplest approach to the problem is to construct a standard VAE, but to add a discretization layer at the end of the encoder, $\boldsymbol{z}_e(\boldsymbol{x}) \in \{0, \ldots, S-1\}^K$, where S is the number of states, and K is the number of discrete latents. For example, we can binarize the latent vector (using $S = 2$) by clipping $\boldsymbol{z}$ to lie in $\{0, 1\}^K$. This can be useful for data compression (see e.g., [BLS17]).

Suppose we assume the prior over the latent codes is uniform. Since the encoder is deterministic, the KL divergence reduces to a constant, equal to $\log K$. This avoids the problem with posterior collapse (Section 21.4). Unfortunately, the discontinuous quantization operation of the encoder prohibits the direct use of gradient based optimization. The solution proposed in [OVK17] is to use the straight-through estimator, which we discuss in Section 6.3.8. We show a simple example of this approach in Figure 21.15, where we use a Gaussian likelihood, so the loss function has the form

$$\mathcal{L} = ||\boldsymbol{x} - d(e(\boldsymbol{x}))||_2^2 \tag{21.75}$$

where $e(\boldsymbol{x}) \in \{0, 1\}^K$ is the encoder, and $d(\boldsymbol{z}) \in \mathbb{R}^{28 \times 28}$ is the decoder.

21.6.2 VQ-VAE model

We can get a more expressive model by using a 3d tensor of discrete latents, $\boldsymbol{z} \in \mathbb{R}^{H \times W \times K}$, where K is the number of discrete values per latent variable. Rather than just binarizing the continuous vector $\boldsymbol{z}_e(\boldsymbol{x})_{ij}$, we compare it to a **codebook** of embedding vectors, $\{\boldsymbol{e}_k : k = 1 : K, \boldsymbol{e}_k \in \mathbb{R}^L\}$, and then set $\boldsymbol{z}_{ij}$ to the index of the nearest codebook entry:

$$q(\boldsymbol{z}_{ij} = k|\boldsymbol{x}) = \begin{cases} 1 & \text{if } k = \text{argmin}_{k'} ||\boldsymbol{z}_e(\boldsymbol{x})_{i,j,:} - \boldsymbol{e}_{k'}||_2 \\ 0 & \text{otherwise} \end{cases} \tag{21.76}$$

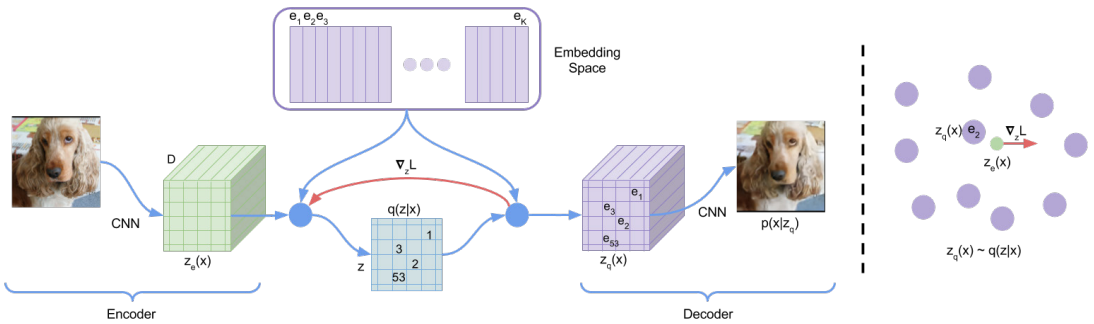


Figure 21.16: VQ-VAE architecture. From Figure 1 of [OVK17]. Used with kind permission of Aäron van den Oord.

When reconstructing the input we replace each discrete code index by the corresponding real-valued codebook vector:

$$(\boldsymbol{z}_q)_{ij} = \boldsymbol{e}_k \text{ where } \boldsymbol{z}_{ij} = k \tag{21.77}$$

These values are then passed to the decoder, $p(\boldsymbol{x}|\boldsymbol{z}_q)$, as usual. See Figure 21.16 for an illustration of the overall architecture. Note that although $\boldsymbol{z}_q$ is generated from a discrete combination of codebook vectors, the use of a distributed code makes the model very expressive. For example, if we use a grid of 32×32, with $K = 512$, then we can generate $512^{32\times 32} = 2^{9216}$ distinct images, which is astronomically large.

To fit this model, we can minimize the negative log likelihood (reconstruction error) using the straight-through estimator, as before. This amounts to passing the gradients from the decoder input $\boldsymbol{z}_q(\boldsymbol{x})$ to the encoder output $\boldsymbol{z}_e(\boldsymbol{x})$, bypassing Equation (21.76), as shown by the red arrow in Figure 21.16. Unfortunately this means that the codebook entries will not get any learning signal. To solve this, the authors proposed to add an extra term to the loss, known as the **codebook loss**, that encourages the codebook entries $\boldsymbol{e}$ to match the output of the encoder. We treat the encoder $\boldsymbol{z}_e(\boldsymbol{x})$ as a fixed target, by adding a **stop gradient** operator to it; this ensures $\boldsymbol{z}_e$ is treated normally in the forwards pass, but has zero gradient in the backwards pass. The modified loss (dropping the spatial indices i, j) becomes

$$\mathcal{L} = -\log p(\boldsymbol{x}|\boldsymbol{z}_q(\boldsymbol{x})) + ||\text{sg}(\boldsymbol{z}_e(\boldsymbol{x})) - \boldsymbol{e}||_2^2 \tag{21.78}$$

where $\boldsymbol{e}$ refers to the codebook vector assigned to $\boldsymbol{z}_e(\boldsymbol{x})$, and sg is the stop gradient operator.

An alternative way to update the codebook vectors is to use moving averages. To see how this works, first consider the batch setting. Let $\{\boldsymbol{z}_{i,1}, \ldots, \boldsymbol{z}_{i,n_i}\}$ be the set of n_i outputs from the encoder that are closest to the dictionary item $\boldsymbol{e}_i$. We can update $\boldsymbol{e}_i$ to minimize the MSE

$$\sum_{j=1}^{n_i} ||\boldsymbol{z}_{i,j} - \boldsymbol{e}_i||_2^2 \tag{21.79}$$

which has the closed form update

$$\boldsymbol{e}_i = \frac{1}{n_i} \sum_{j=1}^{n_i} \boldsymbol{z}_{i,j} \tag{21.80}$$

This is like the M step of the EM algorithm when fitting the mean vectors of a GMM. In the minibatch setting, we replace the above operations with an exponentially moving average, as follows:

$$N_i^t = \gamma N_i^{t-1} + (1-\gamma) n_i^t \tag{21.81}$$

$$\boldsymbol{m}_i^t = \gamma \boldsymbol{m}_i^{t-1} + (1-\gamma) \sum_j \boldsymbol{z}_{i,j}^t \tag{21.82}$$

$$\boldsymbol{e}_i^t = \frac{\boldsymbol{m}_i^t}{N_i^t} \tag{21.83}$$

The authors found $\gamma = 0.9$ to work well.

The above procedure will learn to update the codebook vectors so it matches the output of the encoder. However, it is also important to ensure the encoder does not "change its mind" too often about what codebook value to use. To prevent this, the authors propose to add a third term to the loss, known as the **commitment loss**, that encourages the encoder output to be close to the codebook values. Thus we get the final loss:

$$\mathcal{L} = -\log p(\boldsymbol{x}|\boldsymbol{z}_q(\boldsymbol{x})) + ||\text{sg}(\boldsymbol{z}_e(\boldsymbol{x})) - \boldsymbol{e}||_2^2 + \beta ||\boldsymbol{z}_e(\boldsymbol{x}) - \text{sg}(\boldsymbol{e})||_2^2 \tag{21.84}$$

The authors found $\beta = 0.25$ to work well, although of course the value depends on the scale of the reconstruction loss (NLL) term. (A probabilistic interpretation of this loss can be found in [Hen+18].) Overall, the decoder optimizes the first term only, the encoder optimizes the first and last term, and the embeddings optimize the middle term.

21.6.3 Learning the prior

After training the VQ-VAE model, it is possible to learn a better prior, to match the aggregated posterior. To do this, we just apply the encoder to a set of data, $\{\boldsymbol{x}_n\}$, thus converting them to discrete sequences, $\{\boldsymbol{z}_n\}$. We can then learn a joint distribution $p(\boldsymbol{z})$ using any kind of sequence model. In the original VQ-VAE paper [OVK17], they used the causal convolutional PixelCNN model (Section 22.3.2). More recent work has used transformer decoders (Section 22.4). Samples from this prior can then be decoded using the decoder part of the VQ-VAE model. We give some examples of this in the sections below.

21.6.4 Hierarchical extension (VQ-VAE-2)

In [ROV19], they extend the original VQ-VAE model by using a hierarchical latent code. The model is illustrated in Figure 21.17. They applied this to images of size $256 \times 256 \times 3$. The first latent layer maps this to a quantized representation of size 64×64, and the second latent layer maps this to a quantized representation of size 32×32. This hierarchical scheme allows the top level to focus on high level semantics of the image, leaving fine visual details, such as texture, to the lower level. (See Section 21.5 for more discussion of hierarchical VAEs.)

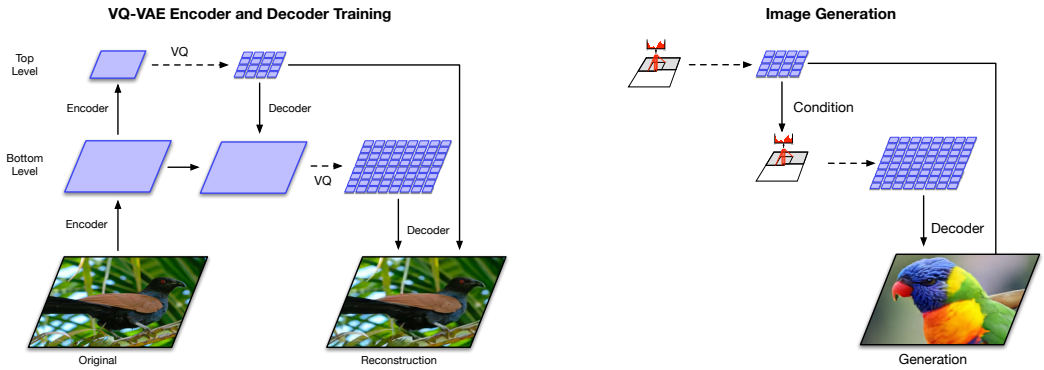


Figure 21.17: Hierarchical extension of VQ-VAE. (a) Encoder and decoder architecture. (b) Combining a Pixel-CNN prior with the decoder. From Figure 2 of [ROV19]. Used with kind permission of Aaron van den Oord.

After fitting the VQ-VAE, they learn a prior over the top level code using a PixelCNN model augmented with self-attention (Section 16.2.7) to capture long-range dependencies. (This hybrid model is known as PixelSNAIL [Che+17c].) For the lower level prior, they just use standard PixelCNN, since attention would be too expensive. Samples from the model can then be decoded using the VQ-VAE decoder, as shown in Figure 21.17.

21.6.5 Discrete VAE

In VQ-VAE, we use a one-hot encoding for the latents, $q(z = k|\boldsymbol{x}) = 1$ iff $k = \text{argmin}_k ||\boldsymbol{z}_e(\boldsymbol{x}) - \boldsymbol{e}_k||_2$, and then set $\boldsymbol{z}_q = \boldsymbol{e}_k$. This does not capture any uncertainty in the latent code, and requires the use of the straight-through estimator for training.

Various other approaches to fitting VAEs with discrete latent codes have been investigated. In the DALL-E paper (Section 22.4.2), they use a fairly simple method, based on using the Gumbel-softmax relaxation for the discrete variables (see Section 6.3.6). In brief, let $q(z = k|\boldsymbol{x})$ be the probability that the input $\boldsymbol{x}$ is assigned to codebook entry k. We can exactly sample $w_k \sim q(z = k|\boldsymbol{x})$ from this by computing $w_k = \text{argmax}_k\, g_k + \log q(z = k|\boldsymbol{x})$, where each g_k is from a Gumbel distribution. We can now "relax" this by using a softmax with temperature $\tau > 0$ and computing

$$w_k = \frac{\exp(\frac{g_k + \log q(z=k|\boldsymbol{x})}{\tau})}{\sum_{j=1}^{K} \exp(\frac{g_j + \log q(z=j|\boldsymbol{x})}{\tau})} \tag{21.85}$$

We now set the latent code to be a weighted sum of the codebook vectors:

$$\boldsymbol{z}_q = \sum_{k=1}^{K} w_k \boldsymbol{e}_k \tag{21.86}$$

In the limit that $\tau \to 0$, the distribution over weights $\boldsymbol{w}$ converges to a one-hot disribution, in which case $\boldsymbol{z}$ becomes equal to one of the codebook entries. But for finite τ, we "fill in" the space between the vectors.

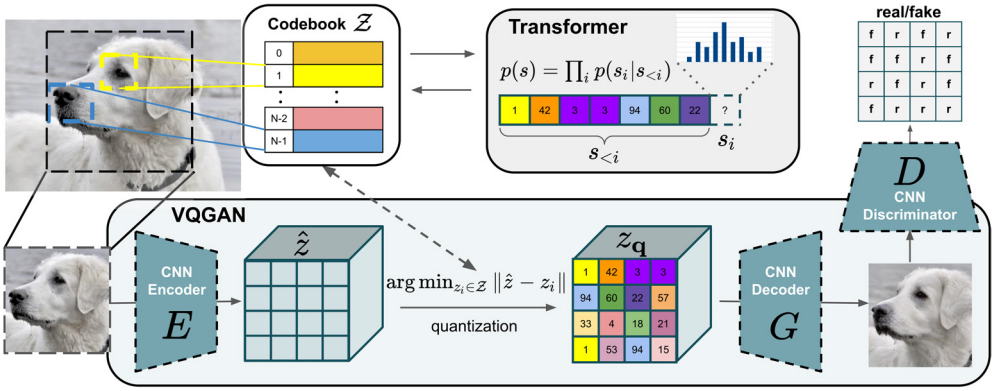


Figure 21.18: Illustration of the VQ-GAN. From Figure 2 of [ERO21]. Used with kind permission of Patrick Esser.

This allows us to express the ELBO in the usual differentiable way:

$$\mathcal{L} = -\mathbb{E}_{q(\boldsymbol{z}|\boldsymbol{x})}\left[\log p(\boldsymbol{x}|\boldsymbol{z})\right] + \beta D_{\mathbb{KL}}\left(q(\boldsymbol{z}|\boldsymbol{x}) \parallel p(\boldsymbol{z})\right) \tag{21.87}$$

where $\beta > 0$ controls the amount of regularization. (Unlike VQ-VAE, the KL term is not a constant, because the encoder is stochastic.) Furthermore, since the Gumbel noise variables are sampled from a distribution that is independent of the encoder parameters, we can use the reparameterization trick (Section 6.3.5) to optimize this.

21.6.6 VQ-GAN

One drawback of VQ-VAE is that it uses mean squared error in its reconstruction loss, which can result in blurry samples. In the **VQ-GAN** paper [ERO21], they replace this with a (patch-wise) GAN loss (see Chapter 26), together with a perceptual loss; this results in much higher visual fidelity. In addition, they use a transformer (see Section 16.3.5) to model the prior on the latent codes. See Figure 21.18 for a visualization of the overall model. In [Yu+21], they replace the CNN encoder and decoder of the VQ-GAN model with transformers, yielding improved results; they call this **VIM** (vector-quantized image modeling).

22 Autoregressive models

22.1 Introduction

By the chain rule of probability, we can write any joint distribution over T variables as follows:

$$p(\boldsymbol{x}_{1:T}) = p(\boldsymbol{x}_1)p(\boldsymbol{x}_2|\boldsymbol{x}_1)p(\boldsymbol{x}_3|\boldsymbol{x}_2,\boldsymbol{x}_1)p(\boldsymbol{x}_4|\boldsymbol{x}_3,\boldsymbol{x}_2,\boldsymbol{x}_1)\ldots = \prod_{t=1}^{T} p(\boldsymbol{x}_t|\boldsymbol{x}_{1:t-1}) \tag{22.1}$$

where $\boldsymbol{x}_t \in \mathcal{X}$ is the t'th observation, and we define $p(\boldsymbol{x}_1|\boldsymbol{x}_{1:0}) = p(x_1)$ as the initial state distribution. This is called an **autoregressive model** or **ARM**. This corresponds to a fully connected DAG, in which each node depends on all its predecessors in the ordering, as shown in Figure 22.1. The models can also be conditioned on arbitrary inputs or context $\boldsymbol{c}$, in order to define $p(\boldsymbol{x}|\boldsymbol{c})$, although we omit this for notational brevity.

We could of course also factorize the joint distribution "backwards" in time, using

$$p(\boldsymbol{x}_{1:T}) = \prod_{t=T}^{1} p(\boldsymbol{x}_t|\boldsymbol{x}_{t+1:T}) \tag{22.2}$$

However, this "anti-causal" direction is often harder to learn (see e.g., [PJS17]).

Although the decomposition in Equation (22.1) is general, each term in this expression (i.e., each conditional distribution $p(\boldsymbol{x}_t|\boldsymbol{x}_{1:t-1})$) becomes more and more complex, since it depends on an increasing number of arguments, which makes the terms slow to compute, and makes estimating their parameters more data hungry (see Section 2.6.3.2).

One approach to solving this intractability is to make the (first-order) **Markov assumption**, which gives rise to a **Markov model** $p(\boldsymbol{x}_t|\boldsymbol{x}_{1:t-1}) = p(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})$, which we discuss in Section 2.6. (This is also called an auto-regressive model of order 1.) Unfortunately, the Markov assumption is very limiting. One way to relax it, and to make $\boldsymbol{x}_t$ depend on all the past $\boldsymbol{x}_{1:t-1}$ without explicitly regressing on them, is to assume the past can be compressed into a **hidden state** $\boldsymbol{z}_t$. If $\boldsymbol{z}_t$ is a deterministic function of the past observations $\boldsymbol{x}_{1:t-1}$, the resulting model is known as a **recurrent neural network**, discussed in Section 16.3.4. If $\boldsymbol{z}_t$ is a stochastic function of the past hidden state, $\boldsymbol{z}_{t-1}$, the resulting model is known as a **hidden Markov model**, which we discuss in Section 29.2.

Another approach is to stay with the general AR model of Equation (22.1), but to use a restricted functional form, such as some kind of neural network, for the conditionals $p(\boldsymbol{x}_t|\boldsymbol{x}_{1:t-1})$. Thus rather than making conditional independence assumptions, or explicitly compressing the past into a sufficient

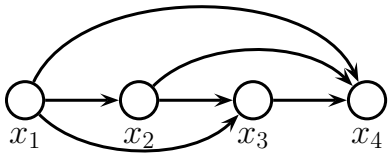


Figure 22.1: A fully-connected auto-regressive model.

statistic, we implicitly learn a compact mapping from the past to the future. In the sections below, we discuss different functional forms for these conditional distributions.

The main advantage of such AR models is that it is easy to compute, and optimize, the exact likelihood of each sequence (data vector). The main disadvantage is that generating samples is inherently sequential, which can be slow. In addition, the method does not learn a compact latent representation of the data.

22.2 Neural autoregressive density estimators (NADE)

A simple way to represent each conditional probability distribution $p(x_t|\boldsymbol{x}_{1:t-1})$ is to use a generalized linear model, such as logistic regression, as proposed in [Fre98]. We can make the model be more powerful by using a neural network. The resulting model is called the **neural auto-regressive density estimator** or **NADE** model [LM11].

If we let $p(x_t|\boldsymbol{x}_{1:t-1})$ be a conditional mixture of Gaussians, we get a model known as **RNADE** ("real-valued neural autoregressive density estimator") of [UML13]. More precisely, this has the form

$$p(x_t|\boldsymbol{x}_{1:t-1}) = \sum_{k=1}^{K} \pi_{t,k} \mathcal{N}(x_t|\mu_{t,k}, \sigma_{t,k}^2) \tag{22.3}$$

where the parameters are generated by a network, $(\boldsymbol{\mu}_t, \boldsymbol{\sigma}_t, \boldsymbol{\pi}_t) = f_t(\boldsymbol{x}_{1:t-1}; \boldsymbol{\theta}_t)$.

Rather than using separate neural networks, $f_1, \ldots, f_T$, it is more efficient to create a single network with T inputs and T outputs. This can be done using masking, resulting in a model called the **MADE** ("masked autoencoder for density estimation") model [Ger+15].

One disadvantage of NADE-type models is that they assume the variables have a natural linear ordering. This makes sense for temporal or sequential data, but not for more general data types, such as images or graphs. An orderless extension to NADE was proposed in [UML14; Uri+16].

22.3 Causal CNNs

One approach to representing the distribution $p(\boldsymbol{x}_t|\boldsymbol{x}_{1:t-1})$ is to try to identify patterns in the past history that might be predictive of the value of $\boldsymbol{x}_t$. If we assume these patterns can occur in any location, it makes sense to use a **convolutional neural network** to detect them. However, we need to make sure we only apply the convolutional mask to past inputs, not future ones. This can be done using **masked convolution**, also called **causal convolution**. We discuss this in more detail below.

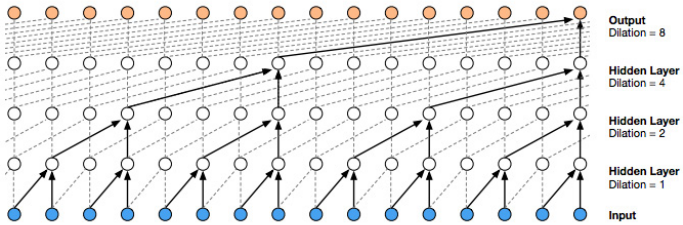


Figure 22.2: Illustration of the wavenet model using dilated (atrous) convolutions, with dilation factors of 1, 2, 4 and 8. From Figure 3 of [oor+16]. Used with kind permission of Aäron van den Oord.

22.3.1 1d causal CNN (convolutional Markov models)

Consider the following **convolutional Markov model** for 1d discrete sequences:

$$p(\boldsymbol{x}_{1:T}) = \prod_{t=1}^{T} p(x_t|\boldsymbol{x}_{1:t-1};\boldsymbol{\theta}) = \prod_{t=1}^{T} \text{Cat}(x_t|\text{softmax}(\varphi(\sum_{\tau=1}^{t-k} \boldsymbol{w}^{\mathsf{T}}\boldsymbol{x}_{\tau:\tau+k}))) \tag{22.4}$$

where $\boldsymbol{w}$ is the convolutional filter of size k, and we have assumed a single nonlinearity φ and categorical output, for notational simplicity. This is like regular 1d convolution except we "mask out" future inputs, so that x_t only depends on the past values. We can of course use deeper models, and we can condition on input features $\boldsymbol{c}$.

In order to capture long-range dependencies, we can use **dilated convolution** (see [Mur22, Sec 14.4.1]). This model has been successfully used to create a state of the art **text to speech** (TTS) synthesis system known as **wavenet** [oor+16]. See Figure 22.2 for an illustration.

The wavenet model is a conditional model, $p(\boldsymbol{x}|\boldsymbol{c})$, where $\boldsymbol{c}$ is a set of linguistic features derived from an input sequence of words, and $\boldsymbol{x}$ is raw audio. The **tacotron** system [Wan+17c] is a fully end-to-end approach, where the input is words rather than linguistic features.

Although wavenet produces high quality speech, it is too slow for use in production systems. However, it can be "distilled" into a parallel generative model [Oor+18], as we discuss in Section 23.2.4.3.

22.3.2 2d causal CNN (PixelCNN)

We can extend causal convolutions to 2d, to get an autoregressive model of the form

$$p(\boldsymbol{x}|\boldsymbol{\theta}) = \prod_{r=1}^{R}\prod_{c=1}^{C} p(x_{r,c}|f_{\boldsymbol{\theta}}(\boldsymbol{x}_{1:r-1,1:C}, \boldsymbol{x}_{r,1:c-1})) \tag{22.5}$$

where R is the number of rows, C is the number of columns, and we condition on all previously generated pixels in a **raster scan** order, as illustrated in Figure 22.3. This is called the **pixelCNN** model [Oor+16]. Naive sampling (generation) from this model takes $O(N)$ time, where $N = RC$ is the number of pixels, but [Ree+17] shows how to use a multiscale approach to reduce the complexity to $O(\log N)$.

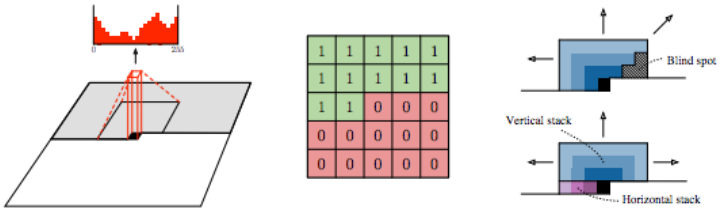


Figure 22.3: Illustration of causal 2d convolution in the PixelCNN model. The red histogram shows the empirical distribution over discretized values for a single pixel of a single RGB channel. The red and green 5×5 array shows the binary mask, which selects the top left context, in order to ensure the convolution is causal. The diagrams on the right illustrate how we can avoid blind spots by using a vertical context stack, that contains all previous rows, and a horizontal context stack, that just contains values from the current row. From Figure 1 of [Oor+16]. Used with kind permission of Aaron van den Oord.

Various extensions of this model have been proposed. The **pixelCNN++** model of [Sal+17c] improved the quality by using a mixture of logistic distributions, to capture the multimodality of $p(x_i|\boldsymbol{x}_{1:i-1})$. The **pixelRNN** of [OKK16] combined masked convolution with an RNN to get even longer range contextual dependencies. The **Subscale Pixel Network** of [MK19] proposed to generate the pixels such that the higher order bits are sampled before lower order bits, which allows high resolution details to be sampled conditioned on low resolution versions of the whole image, rather than just the top left corner.

22.4 Transformers

We introduced transformers in Section 16.3.5. They can be used for encoding sequences (as in BERT), or for decoding (generating) sequences. We can also combine the two, using an encoder-decoder combination, for conditional generation from $p(\boldsymbol{y}|\boldsymbol{c})$. Alternatively, we can define a joint sequence model $p(\boldsymbol{c}, \boldsymbol{y})$, where $\boldsymbol{c}$ is the conditioning or context prompt, and then just condition the joint model, by giving it as the initial context.

The decoder (generator) works as follows. At each step t, the model applies masked (causal) self attention (Section 16.2.7) to the first t inputs, $\boldsymbol{y}_{1:t}$, to compute a set of attention weights, $\boldsymbol{a}_{1:t}$. From this it computes an activation vector $\boldsymbol{z}_t = \sum_{\tau=1}^{t} a_t \boldsymbol{y}_t$. This is then passed through a feed-forward layer to compute $\boldsymbol{h}_t = \text{MLP}(\boldsymbol{z}_t)$. This process is repeated for each layer in the model. Finally the output is used to predict the next element in the sequence, $\boldsymbol{y}_{t+1} \sim \text{Cat}(\text{softmax}(\mathbf{W}\boldsymbol{h}_t))$.

At training time, all predictions can happen in parallel, since the target generated sequence is already available. That is, the t'th output $\boldsymbol{y}_t$ can be predicted given inputs $\boldsymbol{y}_{1:t-1}$, and this can be done for all t simultaneously. However, at test time, the model must be applied sequentially, so the output generated at $t+1$ is fed back into the model to predict $t+2$, etc. Note that the running time of transformers is $O(T^2)$, although a variety of more efficient versions have been developed (see e.g., [Mur22, Sec 15.6] for details).

Transformers are the basis of many popular (conditional) generative models for sequences. We give some examples below.

A "whatpu" is a small, furry animal native to Tanzania. An example of a sentence that uses the word whatpu is:
We were traveling in Africa and we saw these very cute whatpus.

To do a "farduddle" means to jump up and down really fast. An example of a sentence that uses the word farduddle is:
One day when I was playing tag with my little sister, she got really excited and she started doing these crazy farduddles.
A "yalubalu" is a type of vegetable that looks like a big pumpkin. An example of a sentence that uses the word yalubalu is:
I was on a trip to Africa and I tried this yalubalu vegetable that was grown in a garden there. It was delicious.

A "Burringo" is a car with very fast acceleration. An example of a sentence that uses the word Burringo is:
In our garage we have a Burringo that my father drives to work every day.

A "Gigamuru" is a type of Japanese musical instrument. An example of a sentence that uses the word Gigamuru is:
I have a Gigamuru that my uncle gave me as a gift. I love to play it at home.

To "screeg" something is to swing a sword at it. An example of a sentence that uses the word screeg is:
We screeghed at each other for several minutes and then we went outside and ate ice cream.

Figure 22.4: Illustration of few shot learning with GPT-3. The model is asked to create an example sentence using a new word whose meaning is provided in the prompt. Boldface is GPT-3's completions, light gray is human input. From Figure 3.16 of [Bro+20d].

22.4.1 Text generation (GPT, etc.)

In [Rad+18], OpenAI proposed a model called **GPT**, which is short for "Generative Pre-training Transformer". This is a decoder-only transformer model that uses causal (masked) attention. In [Rad+19], they propose **GPT-2**, which is a larger version of GPT (1.5 billion parameters, or 6.5GB, for the XL version), trained on a large web corpus (8 million pages, or 40GB). They also simplify the training objective, and just train it using maximum likelihood.

After GPT-2, OpenAI released **GPT-3** [Bro+20d], which is an even larger version of GPT-2 (175 billion parameters), trained on even more data (300 billion words), but based on the same principles. (Training was estimated to take 355 GPU years and cost $4.6M.) Due to the large size of the data and model, GPT-3 shows even more remarkable abilities to generate novel text. In particular, the output can be (partially) controlled by just changing the conditioning prompt. This enables the model to perform tasks that it has never been trained on, just by giving it some examples in the prompt. This is called "**in-context learning**". See Figure 22.4 for an example, and Section 19.5.1.2 for more details.

More recently, OpenAI released **ChatGPT** [Ope], which is an improved version of GPT-3 which has been trained to have interactive dialogs by using a technique called **reinforcement learning from human feedback** or **RLHF**, a technique first introduced in the **InstructGPT** paper [Ouy+22]. This uses RL techniques such as PPO (Section 35.3.4) to fine tune the model so that it generates responses that are more "aligned" with human intent, as estimated by a ranking model, which is pre-trained on supervised data.

```
(JAX):
Yo, PyTorch, you think you're hot stuff.
But JAX is where it's at, no need to bluff.
Your popularity might be on the rise
But when it comes to performance, we're the guys.

With automatic differentation, we're a breeze
Training neural nets with ease
Our functional approach is innovative and unique
PyTorch can't touch our speed, try and keep up, meek

(PyTorch):
Hold up, JAX,  you might be fast
But PyTorch is the one that's built to last
We may not have the fancy functional flair
But our flexibility sets us above the pair

Our community is strong and growing each day
Suppporting us in every single way
We may not be the newest on the block
But we'lll always be the torch that leads the flock
```

Figure 22.5: ChatGPT response to the prompt "Write a rap battle about PyTorch and JAX". Used with kind permission of Paige Bailey. From https://twitter.com/DynamicWebPaige/status/1601743574369902593.

Despite the impressive performance of these **large language models** or **LLMs** (see Figure 22.5 for an example), there are several open problems with them, such as: they often confidently **hallucinate** incorrect answers to questions (see e.g., [Ji+22]); they can generate biased or toxic output (see e.g., [Lia+]); and they are very resource intensive to train and serve. Indeed, these concerns are why Google has not (at the time of writing) released its version of ChatGPT, known as **LaMDA** [Col21].

The basic ideas behind LLMs are quite simple (maximum likelihood training of an autoregresssive transformer), and they can be implemented in about 300 lines of code.[1] However, just by scaling up the size of the models and datasets, it seems that qualitatively new capabilities can emerge (see e.g., [Wei+22]). Nevertheless, although this approach is good at learning formal linguistic competence (surface form), it is not clear if it is sufficient to learn functional linguistic competence, which requires a deeper, non-linguistic understanding of the world derived from experience [Mah+23].

22.4.2 Image generation (DALL-E, etc.)

The **DALL-E** model[2] from OpenAI [Ram+21a] can generate images of remarkable quality and diversity given text prompts, as shown in Figure 22.6. The methodology is conceptually quite straightforward, and most of the effort went into data collection (they scraped the web for 250 million image-text pairs) and scaling up the training (they fit a model with 12 billion parameters). Here we just focus on the algorithmic methods.

The basic idea is to transform an image $\boldsymbol{x}$ into a sequence of discrete tokens $\boldsymbol{z}$ using a discrete

1. See e.g., https://github.com/karpathy/nanoGPT.
2. The name is derived from the artist Salvador Dalí and Pixar's movied "WALL-E"

(a) an armchair in the shape of an avocado.

(b) an illustration of a baby hedgehog in a christmas sweater walking a dog

Figure 22.6: Some images generated by the DALL-E model in response to a text prompt. (a) "An armchair in the shape of an avocado". (b) "An illustration of a baby hedgehog in a christmas sweater walking a dog". From https://openai.com/blog/dall-e. Used with kind permission of Aditya Ramesh.

VAE model (Section 21.6.5). We then fit a transformer to the concatentation of the image tokens $\boldsymbol{z}$ and text tokens $\boldsymbol{y}$ to get a joint model of the form $p(\boldsymbol{z}, \boldsymbol{y})$.

To sample an image $\boldsymbol{x}$ given a text prompt $\boldsymbol{y}$, we sample a latent code $\boldsymbol{z} \sim p(\boldsymbol{z}|\boldsymbol{y})$ by conditioning the transformer on the prompt $\boldsymbol{y}$, and then we feed $\boldsymbol{z}$ into the VAE decoder to get the image $\boldsymbol{x} \sim p(\boldsymbol{x}|\boldsymbol{z})$. Multiple images are generated for each prompt, and these are then ranked according to a pre-trained critic, which gives them scores depending on how well the generated image matches the input text: $s_n = \text{critic}(\boldsymbol{x}_n, \boldsymbol{y}_n)$. The critic they used was the contrastive CLIP model (see Section 32.3.4.1). This discriminative reranking significantly improves the results.

Some sample results are shown in Figure 22.6, and more can be found online at https://openai.com/blog/dall-e/. The image on the right of Figure 22.6 is particularly interesting, since the prompt — "An illustration of a baby hedgehog in a christmas sweater walking a dog" — arguably requires that the model solve the "**variable binding problem**". This refers to the fact that the sentence implies the hedgehog should be wearing the sweater and not the dog. We see that the model sometimes interprets this correctly, but not always: sometimes it draws both animals with Christmas sweaters. In addition, sometimes it draws a hedgehog walking a smaller hedgehog. The quality of the results can also be sensitive to the form of the prompt.

The **PARTI** model [Yu+22] from Google follows similar high level ideas to DALL-E, but has been scaled to an even larger size. The larger models perform qualitatively much better, as shown in Figure 20.3.

Other recent approaches to (conditional) image generation — such as **DALL-E 2** [Ram+22] from Open-AI, **Imagen** [Sah+22b] from Google, and **Stable diffusion** [Rom+22] from Stability.AI — are based on diffusion rather than applying a transformer to discretized image patches. See Section 25.6.4 for details.

22.4.3 Other applications

Transformers have been used to generate many other kinds of (discrete) data, such as midi music sequences [Hua+18a], protein sequences [Gan+23], etc.

23 Normalizing flows

This chapter is written by George Papamakarios and Balaji Lakshminarayanan.

23.1 Introduction

In this chapter we discuss **normalizing flows**, a class of flexible density models that can be easily sampled from and whose exact likelihood function is efficient to compute. Such models can be used for many tasks, such as density modeling, inference and generative modeling. We introduce the key principles of normalizing flows and refer to recent surveys by Papamakarios et al. [Pap+19] and Kobyzev, Prince, and Brubaker [KPB19] for readers interested in learning more. See also https://github.com/janosh/awesome-normalizing-flows for a list of papers and software packages.

23.1.1 Preliminaries

Normalizing flows create complex probability distributions $p(\boldsymbol{x})$ by passing random variables $\boldsymbol{u} \in \mathbb{R}^D$, drawn from a simple **base distribution** $p(\boldsymbol{u})$ through a nonlinear but *invertible* transformation $\boldsymbol{f} : \mathbb{R}^D \to \mathbb{R}^D$. That is, $p(\boldsymbol{x})$ is defined by the following process:

$$\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{u}) \quad \text{where} \quad \boldsymbol{u} \sim p(\boldsymbol{u}). \tag{23.1}$$

The base distribution is typically chosen to be simple, for example standard Gaussian or uniform, so that we can easily sample from it and compute the density $p(\boldsymbol{u})$. A flexible enough transformation $\boldsymbol{f}$ can induce a complex distribution on the transformed variable $\boldsymbol{x}$ even if the base distribution is simple.

Sampling from $p(\boldsymbol{x})$ is straightforward: we first sample $\boldsymbol{u}$ from $p(\boldsymbol{u})$ and then compute $\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{u})$. To compute the density $p(\boldsymbol{x})$, we rely on the fact that $\boldsymbol{f}$ is invertible. Let $\boldsymbol{g}(\boldsymbol{x}) = \boldsymbol{f}^{-1}(\boldsymbol{x}) = \boldsymbol{u}$ be the inverse mapping, which "**normalizes**" the data distribution by mapping it back to the base distribution (which is often a normal distribution). Using the change-of-variables formula for random variables from Equation (2.257), we have

$$p_x(\boldsymbol{x}) = p_u(\boldsymbol{g}(\boldsymbol{x}))|\det \mathbf{J}(\boldsymbol{g})(\boldsymbol{x})| = p_u(\boldsymbol{u})|\det \mathbf{J}(\boldsymbol{f})(\boldsymbol{u})|^{-1}, \tag{23.2}$$

where $\mathbf{J}(\boldsymbol{f})(\boldsymbol{u}) = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}}|_{\boldsymbol{u}}$ is the Jacobian matrix of $\boldsymbol{f}$ evaluated at $\boldsymbol{u}$. Taking logs of both sides of Equation (23.2), we get

$$\log p_x(\boldsymbol{x}) = \log p_u(\boldsymbol{u}) - \log|\det \mathbf{J}(\boldsymbol{f})(\boldsymbol{u})|. \tag{23.3}$$

As discussed above, $p(\boldsymbol{u})$ is typically easy to evaluate. So, if one can use flexible invertible transformations $\boldsymbol{f}$ whose Jacobian determinant $\det \mathbf{J}(\boldsymbol{f})(\boldsymbol{u})$ can be computed efficiently, then one can construct complex densities $p(\boldsymbol{x})$ that allow exact sampling and efficient exact likelihood computation. This is in contrast to latent variable models, which require methods like variational inference to lower-bound the likelihood.

One might wonder how flexible are the densities $p(\boldsymbol{x})$ obtained by transforming random variables sampled from simple $p(\boldsymbol{u})$. It turns out that we can use this method to approximate any smooth distribution. To see this, consider the scenario where the base distribution $p(\boldsymbol{u})$ is a one-dimensional uniform distribution. Recall that inverse transform sampling (Section 11.3.1) samples random variables from a uniform distribution and transforms them using the inverse cumulative distribution function (cdf) to generate samples from the desired density. We can use this method to sample from any one-dimensional density as long as the transformation $\boldsymbol{f}$ is powerful enough to model the inverse cdf (which is a reasonable assumption for well-behaved densities whose cdf is invertible and differentiable). We can further extend this argument to multiple dimensions by first expressing the density $p(\boldsymbol{x})$ as a product of one-dimensional conditionals using the chain rule of probability, and then applying inverse transform sampling to each one-dimensional conditional. The result is a normalizing flow that transforms a product of uniform distributions into any desired distribution $p(\boldsymbol{x})$. We refer to [Pap+19] for a more detailed proof.

How do we define flexible invertible mappings whose Jacobian determinant is easy to compute? We discuss this topic in detail in Section 23.2, but in summary, there are two main ways. The first approach is to define a set of simple transformations that are invertible by design, and whose Jacobian determinant is easy to compute; for instance, if the Jacobian is a triangular matrix, its determinant can be computed efficiently. The second approach is to exploit the fact that a composition of invertible functions is also invertible, and the overall Jacobian determinant is just the product of the individual Jacobian determinants. More precisely, if $\boldsymbol{f} = \boldsymbol{f}_N \circ \cdots \circ \boldsymbol{f}_1$ where each $\boldsymbol{f}_i$ is invertible, then $\boldsymbol{f}$ is also invertible, with inverse $\boldsymbol{g} = \boldsymbol{g}_1 \circ \cdots \circ \boldsymbol{g}_N$ and log Jacobian determinant given by

$$\log |\det \mathbf{J}(\boldsymbol{g})(\boldsymbol{x})| = \sum_{i=1}^{N} \log |\det \mathbf{J}(\boldsymbol{g}_i)(\boldsymbol{u}_i)| \tag{23.4}$$

where $\boldsymbol{u}_i = \boldsymbol{f}_i \circ \cdots \circ \boldsymbol{f}_1(\boldsymbol{u})$ is the i'th intermediate output of the flow. This allows us to create complex flows from simple components, just as graphical models allow us to create complex joint distributions from simpler conditional distributions.

Finally, a note on terminology. An invertible transformation is also known as a **bijection**. A bijection that is differentiable and has a differentiable inverse is known as a **diffeomorphism**. The transformation $\boldsymbol{f}$ of a flow model is a diffeomorphism, although in the rest of this chapter we will refer to it as a "bijection" for simplicity, leaving the differentiability implicit. The density $p_x(\boldsymbol{x})$ of a flow model is also known as the **pushforward** of the base distribution $p_u(\boldsymbol{u})$ through the transformation $\boldsymbol{f}$, and is sometimes denoted as $p_x = \boldsymbol{f}_* p_u$. Finally, in mathematics the term "flow" refers to any family of diffeomorphisms $\boldsymbol{f}_t$ indexed by a real number t such that $t = 0$ indexes the identity function, and $t_1 + t_2$ indexes $\boldsymbol{f}_{t_2} \circ \boldsymbol{f}_{t_1}$ (in physics, t often represents time). In machine learning we use the term "flow" by analogy to the above meaning, to highlight the fact that we can create flexible invertible transformations by composing simpler ones; in this sense, the index t is analogous to the number i of transformations in $\boldsymbol{f}_i \circ \cdots \circ \boldsymbol{f}_1$.

23.1.2 How to train a flow model

There are two common applications of normalizing flows. The first one is density estimation of observed data, which is achieved by fitting $p_{\boldsymbol{\theta}}(\boldsymbol{x})$ to the data and using it as an estimate of the data density, potentially followed by generating new data from $p_{\boldsymbol{\theta}}(\boldsymbol{x})$. The second one is variational inference, which involves sampling from and evaluating a variational posterior $q_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})$ parameterized by the flow model. As we will see below, these applications optimize different objectives and impose different computational constraints on the flow model.

23.1.2.1 Density estimation

Density estimation requires maximizing the likelihood function in Equation (23.2). This requires that we can efficiently evaluate the inverse flow $\boldsymbol{u} = \boldsymbol{f}^{-1}(\boldsymbol{x})$ and its Jacobian determinant $\det \mathbf{J}(\boldsymbol{f}^{-1})(\boldsymbol{x})$ for any given $\boldsymbol{x}$. After optimizing the model, we can optionally use it to generate new data. To sample new points, we require that the forwards mapping $\boldsymbol{f}$ be tractable.

23.1.2.2 Variational inference

Normalizing flows are commonly used for variational inference to parameterize the approximate posterior distribution in latent variable models, as discussed in Section 10.4.3. Consider a latent variable model with continuous latent variables $\boldsymbol{z}$ and observable variables $\boldsymbol{x}$. For simplicity, we consider the model parameters to be fixed as we are interested in approximating the true posterior $p^*(\boldsymbol{z}|\boldsymbol{x})$ with a normalizing flow $q_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})$.[1] As discussed in Section 10.1.1.2, the variational parameters are trained by maximizing the evidence lower bound (ELBO), given by

$$L(\boldsymbol{\theta}) = \mathbb{E}_{q_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})} \left[\log p(\boldsymbol{x}|\boldsymbol{z}) + \log p(\boldsymbol{z}) - \log q_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})\right] \tag{23.5}$$

When viewing the ELBO as a function of $\boldsymbol{\theta}$, it can be simplified as follows (note we drop the dependency on $\boldsymbol{x}$ for simplicity):

$$L(\boldsymbol{\theta}) = \mathbb{E}_{q_{\boldsymbol{\theta}}(\boldsymbol{z})} \left[\ell_{\boldsymbol{\theta}}(\boldsymbol{z})\right]. \tag{23.6}$$

Let $q_{\boldsymbol{\theta}}(\boldsymbol{z})$ denote a normalizing flow with base distribution $q(\boldsymbol{u})$ and transformation $\boldsymbol{z} = f_{\boldsymbol{\theta}}(\boldsymbol{u})$. Then the reparameterization trick (Section 6.3.5) allows us to optimize the parameters using stochastic gradients. To achieve this, we first write the expectation with respect to the base distribution:

$$L(\boldsymbol{\theta}) = \mathbb{E}_{q_{\boldsymbol{\theta}}(\boldsymbol{z})} \left[\ell_{\boldsymbol{\theta}}(\boldsymbol{z})\right] = \mathbb{E}_{q(\boldsymbol{u})} \left[\ell_{\boldsymbol{\theta}}(f_{\boldsymbol{\theta}}(\boldsymbol{u}))\right]. \tag{23.7}$$

Then, since the base distribution does not depend on $\boldsymbol{\theta}$, we can obtain stochastic gradients as follows:

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) = \mathbb{E}_{q(\boldsymbol{u})} \left[\nabla_{\boldsymbol{\theta}} \ell_{\boldsymbol{\theta}}(f_{\boldsymbol{\theta}}(\boldsymbol{u}))\right] \approx \frac{1}{N} \sum_{n=1}^{N} \nabla_{\boldsymbol{\theta}} \ell_{\boldsymbol{\theta}}(f_{\boldsymbol{\theta}}(\boldsymbol{u}_n)), \tag{23.8}$$

where $\{\boldsymbol{u}_n\}_{n=1}^N$ are samples from $q(\boldsymbol{u})$.

1. We denote the parameters of the variational posterior by $\boldsymbol{\theta}$ here, which should not be confused with the model parameters which are also typically denoted by $\boldsymbol{\theta}$ elsewhere.

As we can see, in order to optimize this objective, we need to be able to efficiently sample from $q_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})$ and evaluate the probability density of these samples during optimization. (See Section 23.2.4.3 for details on how to do this.) This is contrast to the MLE approach in Section 23.1.2.1, which requires that we be able to compute efficiently the density of arbitrary training datapoints, but it does not require samples during optimization.

23.2 Constructing flows

In this section, we discuss how to compute various kinds of flows that are invertible by design and have efficiently computable Jacobian determinants.

23.2.1 Affine flows

A simple choice is to use an affine transformation $\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{u}) = \mathbf{A}\boldsymbol{u} + \boldsymbol{b}$. This is a bijection if and only if $\mathbf{A}$ is an invertible square matrix. The Jacobian determinant of $\boldsymbol{f}$ is $\det \mathbf{A}$, and its inverse is $\boldsymbol{u} = \boldsymbol{f}^{-1}(\boldsymbol{x}) = \mathbf{A}^{-1}(\boldsymbol{x} - \boldsymbol{b})$. A flow consisting of affine bijections is called an **affine flow**, or a **linear flow** if we ignore $\boldsymbol{b}$.

On their own, affine flows are limited in their expressive power. For example, suppose the base distribution is Gaussian, $p(\boldsymbol{u}) = \mathcal{N}(\boldsymbol{u}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$. Then the pushforward distribution after an affine bijection is still Gaussian, $p(\boldsymbol{x}) = \mathcal{N}(\boldsymbol{x}|\mathbf{A}\boldsymbol{\mu}+\boldsymbol{b}, \mathbf{A}\boldsymbol{\Sigma}\mathbf{A}^{\mathsf{T}})$. However, affine bijections are useful building blocks when composed with the non-affine bijections we discuss later, as they encourage "mixing" of dimensions through the flow.

For practical reasons, we need to ensure the Jacobian determinant and the inverse of the flow are fast to compute. In general, computing $\det \mathbf{A}$ and $\mathbf{A}^{-1}$ explicitly takes $O(D^3)$ time. To reduce the cost, we can add structure to $\mathbf{A}$. If $\mathbf{A}$ is diagonal, the cost becomes $O(D)$. If $\mathbf{A}$ is triangular, the Jacobian determinant is the product of the diagonal elements, so it takes $O(D)$ time; inverting the flow requires solving the triangular system $\mathbf{A}\boldsymbol{u} = \boldsymbol{x} - \boldsymbol{b}$, which can be done with backsubstitution in $O(D^2)$ time.

The result of a triangular transformation depends on the ordering of the dimensions. To reduce sensitivity to this, and to encourage "mixing" of dimensions, we can multiply $\mathbf{A}$ with a permutation matrix, which has an absolute determinant of 1. We often use a permutation that reverses the indices at each layer or that randomly shuffles them. However, usually the permutation at each layer is fixed rather than learned.

For spatially structured data (such as images), we can define $\mathbf{A}$ to be a convolution matrix. For example, GLOW [KD18b] uses 1×1 convolution; this is equivalent to pointwise linear transformation across feature dimensions, but regular convolution across spatial dimensions. Two more general methods for modeling $d \times d$ convolutions are presented in [HBW19], one based on stacking autoregressive convolutions, and the other on carrying out the convolution in the Fourier domain.

23.2.2 Elementwise flows

Let $h : \mathbb{R} \to \mathbb{R}$ be a scalar-valued bijection. We can create a vector-valued bijection $\boldsymbol{f} : \mathbb{R}^D \to \mathbb{R}^D$ by applying h elementwise, that is, $\boldsymbol{f}(\boldsymbol{u}) = (h(u_1), \ldots, h(u_D))$. The function $\boldsymbol{f}$ is invertible, and its Jacobian determinant is given by $\prod_{i=1}^{D} \frac{dh}{du_i}$. A flow composed of such bijections is known as an **elementwise flow**.

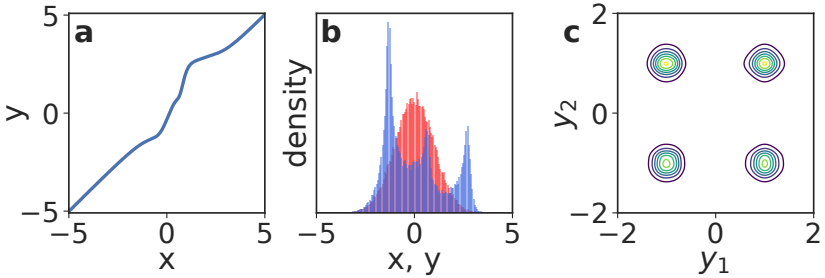


Figure 23.1: Non-linear squared flow (NLSq). Left: an invertible mapping consisting of 4 NLSq layers. Middle: red is the base distribution (Gaussian), blue is the distribution induced by the mapping on the left. Right: density of a 5-layer autoregressive flow using NLSq transformations and a Gaussian base density, trained on a mixture of 4 Gaussians. From Figure 5 of [ZR19b]. Used with kind permission of Zachary Ziegler.

On their own, elementwise flows are limited, since they do not model dependencies between the elements. However, they are useful building blocks for more complex flows, such as coupling flows (Section 23.2.3) and autoregressive flows (Section 23.2.4), as we will see later. In this section, we discuss techniques for constructing scalar-valued bijections $h : \mathbb{R} \to \mathbb{R}$ for use in elementwise flows.

23.2.2.1 Affine scalar bijection

An **affine scalar bijection** has the form $h(u; \boldsymbol{\theta}) = au + b$, where $\boldsymbol{\theta} = (a, b) \in \mathbb{R}^2$. (This is a scalar version of an affine flow.) Its derivative $\frac{dh}{du}$ is equal to a. It is invertible if and only if $a \neq 0$. In practice, we often parameterize a to be positive, for example by making it the exponential or the softplus of an unconstrained parameter. When $a = 1$, $h(u; \boldsymbol{\theta}) = u + b$ is often called an **additive scalar bijection**.

23.2.2.2 Higher-order perturbations

The affine scalar bijection is simple to use, but limited. We can make it more flexible by adding higher-order perturbations, under the constraint that invertibility is preserved. For example, Ziegler and Rush [ZR19b] propose the following, which they term **non-linear squared flow**:

$$h(u; \boldsymbol{\theta}) = au + b + \frac{c}{1 + (du + e)^2}, \tag{23.9}$$

where $\boldsymbol{\theta} = (a, b, c, d, e) \in \mathbb{R}^5$. When $c = 0$, this reduces to the affine case. When $c \neq 0$, it adds an inverse-quadratic perturbation, which can induce multimodality as shown in Figure 23.1. Under the constraints $a > \frac{9}{8\sqrt{3}}cd$ and $d > 0$ the function becomes invertible, and its inverse can be computed analytically by solving a quadratic polynomial.

23.2.2.3 Combinations of strictly monotonic scalar functions

A strictly monotonic scalar function is one that is always increasing (has positive derivative everywhere) or always decreasing (has negative derivative everywhere). Such functions are invertible. Many

activation functions, such as the logistic sigmoid $\sigma(u) = 1/(1 + \exp(-u))$, are strictly monotonic.

Using such activation functions as a starting point, we can build more flexible monotonic functions via **conical combination** (linear combination with positive coefficients) and function composition. Suppose $h_1, \ldots, h_K$ are strictly increasing; then the following are also strictly increasing:

- $a_1 h_1 + \cdots + a_K h_K + b$ with $a_k > 0$ (conical combination with a bias),
- $h_1 \circ \cdots \circ h_K$ (function composition).

By repeating the above two constructions, we can build arbitrarily complex increasing functions. For example, a composition of conical combinations of logistic sigmoids is just an MLP where all weights are positive [Hua+18b].

The derivative of such a scalar bijection can be computed by repeatedly applying the chain rule, and in practice can be done with automatic differentiation. However, the inverse is not typically computable in closed form. In practice we can compute the inverse using bisection search, since the function is monotonic.

23.2.2.4 Scalar bijections from integration

A simple way to ensure a scalar function is strictly monotonic is to constrain its derivative to be positive. Let $h' = \frac{dh}{du}$ be this derivative. Wehenkel and Louppe [WL19] directly parameterize h' with a neural network whose output is made positive via an ELU activation function shifted up by 1. They then integrate the derivative numerically to get the bijection:

$$h(u) = \int_0^u h'(t)dt + b, \tag{23.10}$$

where b is a bias. They call this approach **unconstrained monotonic neural networks**.

The above integral is generally not computable in closed form. It can be, however, if h' is constrained appropriately. For example, Jaini, Selby, and Yu [JSY19] take h' to be a sum of K squared polynomials of degree L:

$$h'(u) = \sum_{k=1}^{K} \left(\sum_{\ell=0}^{L} a_{k\ell}\, u^\ell \right)^2 . \tag{23.11}$$

This makes h' a non-negative polynomial of degree $2L$. The integral is analytically tractable, and makes h an increasing polynomial of degree $2L + 1$. For $L = 0$, h' is constant, so h reduces to an affine scalar bijection.

In these approaches, the derivative of the bijection can just be read off. However, the inverse is not analytically computable in general. In practice, we can use bisection search to compute the inverse numerically.

23.2.2.5 Splines

Another way to construct monotonic scalar functions is using **splines**. These are piecewise-polynomial or piecewise-rational functions, parameterized in terms of $K + 1$ **knots** (u_k, x_k) through which the spline passes. That is, we set $h(u_k) = x_k$, and define h on the interval (u_{k-1}, u_k) by interpolating

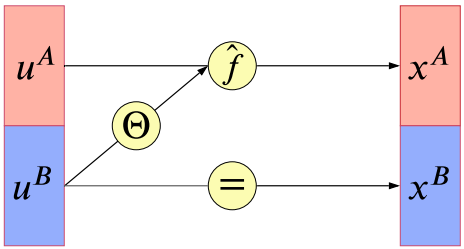


Figure 23.2: Illustration of a coupling layer $\boldsymbol{x} = f(\boldsymbol{u})$. A bijection, with parameters determined by $\boldsymbol{u}^B$, is applied to $\boldsymbol{u}^A$ to generate $\boldsymbol{x}^A$; meanwhile $\boldsymbol{x}^B = \boldsymbol{u}^B$ is passed through unchanged, so the mapping can be inverted. From Figure 3 of [KPB19]. Used with kind permission of Ivan Kobyzev.

from x_{k-1} to x_k with a polynomial or rational function (ratio of two polynomials). By increasing the number of knots we can create arbitrarily flexible monotonic functions.

Different ways to interpolate between knots give different types of spline. A simple choice is to interpolate linearly [Mül+19a]. However, this makes the derivative discontinuous at the knots. Interpolating with quadratic polynomials [Mül+19a] gives enough flexibility to make the derivative continuous. Interpolating with cubic polynomials [Dur+19], ratios of linear polynomials [DEL20], or ratios of quadratic polynomials [DBP19] allows the derivatives at the knots to be arbitrary parameters.

The spline is strictly increasing if we take $u_{k-1} < u_k$, $x_{k-1} < x_k$, and make sure the interpolation between knots is itself increasing. Depending on the flexibility on the interpolating function, more than one interpolation may exist; in practice we choose one that is guaranteed to be always increasing (see references above for details).

An advantage of splines is that they can be inverted analytically if the interpolating functions only contain low-degree polynomials. In this case, we compute $u = h^{-1}(x)$ as follows: first, we use binary search to locate the interval (x_{k-1}, x_k) in which x lies; then, we analytically solve the resulting low-degree polynomial for u.

23.2.3 Coupling flows

In this section we describe coupling flows, which allow us to model dependencies between dimensions using arbitrary non-linear functions (such as deep neural networks). Consider a partition of the input $\boldsymbol{u} \in \mathbb{R}^D$ into two subspaces, $(\boldsymbol{u}^A, \boldsymbol{u}^B) \in \mathbb{R}^d \times \mathbb{R}^{D-d}$, where d is an integer between 1 and $D-1$. Assume a bijection $\hat{\mathbf{f}}(\cdot; \boldsymbol{\theta}) : \mathbb{R}^d \to \mathbb{R}^d$ parameterized by $\boldsymbol{\theta}$ and acting on the subspace $\mathbb{R}^d$. We define the function $\boldsymbol{f} : \mathbb{R}^D \to \mathbb{R}^D$ given by $\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{u})$ as follows:

$$\boldsymbol{x}^A = \hat{\mathbf{f}}(\boldsymbol{u}^A; \Theta(\boldsymbol{u}^B)) \tag{23.12}$$

$$\boldsymbol{x}^B = \boldsymbol{u}^B. \tag{23.13}$$

See Figure 23.2 for an illustration. The function $\boldsymbol{f}$ is called a **coupling layer** [DKB15; DSDB17], because it "couples" $\boldsymbol{u}^A$ and $\boldsymbol{u}^B$ together though $\hat{\mathbf{f}}$ and Θ. We refer to flows consisting of coupling layers as **coupling flows**.

The parameters of $\hat{\mathbf{f}}$ are computed by $\boldsymbol{\theta} = \Theta(\boldsymbol{u}^B)$, where Θ is an *arbitrary* function called the **conditioner**. Unlike affine flows, which mix dimensions linearly, and elementwise flows, which do not mix dimensions at all, coupling flows can mix dimensions with a flexible non-linear conditioner Θ. In practice we often implement Θ as a deep neural network; any architecture can be used, including MLPs, CNNs, ResNets, etc.

The coupling layer $\boldsymbol{f}$ is *invertible*, and its inverse is given by $\boldsymbol{u} = \boldsymbol{f}^{-1}(\boldsymbol{x})$, where

$$\boldsymbol{u}^A = \hat{\mathbf{f}}^{-1}(\boldsymbol{x}^A; \Theta(\boldsymbol{x}^B)) \tag{23.14}$$

$$\boldsymbol{u}^B = \boldsymbol{x}^B. \tag{23.15}$$

That is, $\boldsymbol{f}^{-1}$ is given by simply replacing $\hat{\mathbf{f}}$ with $\hat{\mathbf{f}}^{-1}$. Because $\boldsymbol{x}^B$ does not depend on $\boldsymbol{u}^A$, the Jacobian of $\boldsymbol{f}$ is block triangular:

$$\mathbf{J}(\boldsymbol{f}) = \begin{pmatrix} \partial \boldsymbol{x}^A / \partial \boldsymbol{u}^A & \partial \boldsymbol{x}^A / \partial \boldsymbol{u}^B \\ \partial \boldsymbol{x}^B / \partial \boldsymbol{u}^A & \partial \boldsymbol{x}^B / \partial \boldsymbol{u}^B \end{pmatrix} = \begin{pmatrix} \mathbf{J}(\hat{\mathbf{f}}) & \partial \boldsymbol{x}^A / \partial \boldsymbol{u}^B \\ \mathbf{0} & \mathbf{I} \end{pmatrix}. \tag{23.16}$$

Thus, $\det \mathbf{J}(\boldsymbol{f})$ is equal to $\det \mathbf{J}(\hat{\mathbf{f}})$.

We often define $\hat{\mathbf{f}}$ to be an elementwise bijection, so that $\hat{\mathbf{f}}^{-1}$ and $\det \mathbf{J}(\hat{\mathbf{f}})$ are easy to compute. That is, we define:

$$\hat{\mathbf{f}}(\boldsymbol{u}^A; \boldsymbol{\theta}) = \left(h(u_1^A; \boldsymbol{\theta}_1), \ldots, h(u_d^A; \boldsymbol{\theta}_d)\right), \tag{23.17}$$

where $h(\cdot; \boldsymbol{\theta}_i)$ is a scalar bijection parameterized by $\boldsymbol{\theta}_i$. Any of the scalar bijections described in Section 23.2.2 can be used here. For example, $h(\cdot; \boldsymbol{\theta}_i)$ can be an affine bijection with $\boldsymbol{\theta}_i$ its scale and shift parameters (Section 23.2.2.1); or it can be a monotonic MLP with $\boldsymbol{\theta}_i$ its weights and biases (Section 23.2.2.3); or it can be a monotonic spline with $\boldsymbol{\theta}_i$ its knot coordinates (Section 23.2.2.5).

There are many ways to define the partition of $\boldsymbol{u}$ into $(\boldsymbol{u}^A, \boldsymbol{u}^B)$. A simple way is just to partition $\boldsymbol{u}$ into two halves. We can also exploit spatial structure in the partitioning. For example, if $\boldsymbol{u}$ is an image, we can partition its pixels using a "checkerboard" pattern, where pixels in "black squares" are in $\boldsymbol{u}^A$ and pixels in "white squares" are in $\boldsymbol{u}^B$ [DSDB17]. Since only part of the input is transformed by each coupling layer, in practice we typically employ different partitions along a coupling flow, to ensure all variables get transformed and are given the opportunity to interact.

Finally, if $\hat{\mathbf{f}}$ is an elementwise bijection, we can implement arbitrary partitions easily using a binary mask $\boldsymbol{b}$ as follows:

$$\boldsymbol{x} = \boldsymbol{b} \odot \boldsymbol{u} + (1 - \boldsymbol{b}) \odot \hat{\mathbf{f}}(\boldsymbol{u}; \Theta(\boldsymbol{b} \odot \boldsymbol{u})), \tag{23.18}$$

where $\odot$ denotes elementwise multiplication. A value of 0 in $\boldsymbol{b}$ indicates that the corresponding element in $\boldsymbol{u}$ is transformed (belongs to $\boldsymbol{u}^A$); a value of 1 indicates that it remains unchanged (belongs to $\boldsymbol{u}^B$).

As an example, we fit a masked coupling flow, created from piecewise rational quadratic splines, to the two moons dataset. Samples from each layer of the fitted model are shown in Figure 23.3.

23.2.4 Autoregressive flows

In this section we discuss **autoregressive flows**, which are flows composed of autoregressive bijections. Like coupling flows, autoregressive flows allow us to model dependencies between variables with arbitrary non-linear functions, such as deep neural networks.

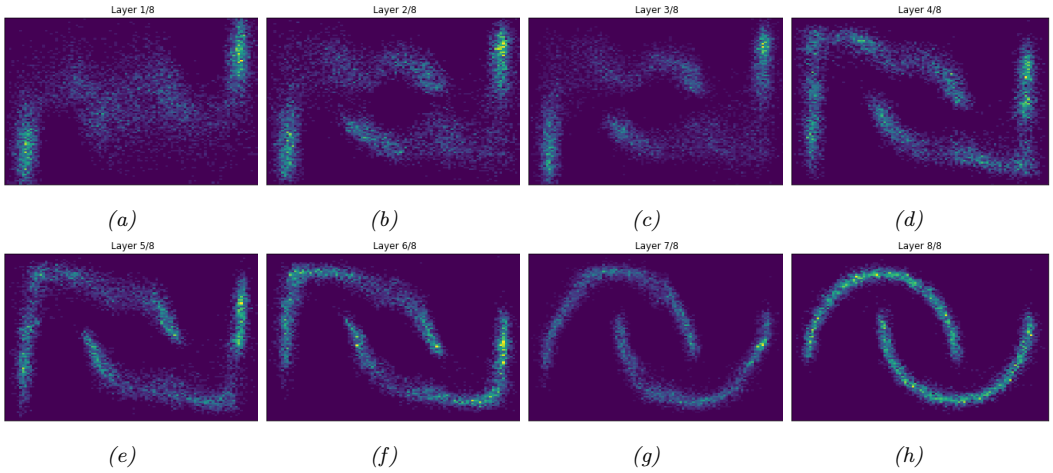


Figure 23.3: (a) Two moons dataset. (b) Samples from a normalizing flow fit to this dataset. Generated by two_moons_nsf_normalizing_flow.ipynb.

Suppose the input $\boldsymbol{u}$ contains D scalar elements, that is, $\boldsymbol{u} = (u_1, \ldots, u_D) \in \mathbb{R}^D$. We define an **autoregressive bijection** $\boldsymbol{f} : \mathbb{R}^D \to \mathbb{R}^D$, its output denoted by $\boldsymbol{x} = (x_1, \ldots, x_D) \in \mathbb{R}^D$, as follows:

$$x_i = h(u_i; \Theta_i(\boldsymbol{x}_{1:i-1})), \quad i = 1, \ldots, D. \tag{23.19}$$

Each output x_i depends on the corresponding input u_i and all previous outputs $\boldsymbol{x}_{1:i-1} = (x_1, \ldots, x_{i-1})$. The function $h(\cdot; \boldsymbol{\theta}) : \mathbb{R} \to \mathbb{R}$ is a scalar bijection (for example, one of those described in Section 23.2.2), and is parameterized by $\boldsymbol{\theta}$. The function Θ_i is a conditioner that outputs the parameters $\boldsymbol{\theta}_i$ that yield x_i, given all previous outputs $\boldsymbol{x}_{1:i-1}$. Like in coupling flows, Θ_i can be an arbitrary non-linear function, and is often parameterized as a deep neural network.

Because h is invertible, $\boldsymbol{f}$ is also invertible, and its inverse is given by:

$$u_i = h^{-1}(x_i; \Theta_i(\boldsymbol{x}_{1:i-1})), \quad i = 1, \ldots, D. \tag{23.20}$$

An important property of $\boldsymbol{f}$ is that each output $\boldsymbol{x}_i$ depends on $\boldsymbol{u}_{1:i} = (u_1, \ldots, u_i)$, but not on $\boldsymbol{u}_{i+1:D} = (u_{i+1}, \ldots, u_D)$; as a result, the partial derivative $\partial x_i / \partial u_j$ is identically zero whenever $j > i$. Therefore, the Jacobian matrix $\mathbf{J}(\boldsymbol{f})$ is triangular, and its determinant is simply the product of its diagonal entries:

$$\det \mathbf{J}(\boldsymbol{f}) = \prod_{i=1}^{D} \frac{\partial x_i}{\partial u_i} = \prod_{i=1}^{D} \frac{dh}{du_i}. \tag{23.21}$$

In other words, the autoregressive structure of $\boldsymbol{f}$ leads to a Jacobian determinant that can be computed efficiently in $O(D)$ time.

Although invertible, autoregressive bijections are computationally asymmetric: evaluating $\boldsymbol{f}$ is inherently sequential, whereas evaluating $\boldsymbol{f}^{-1}$ is inherently parallel. That is because we need $\boldsymbol{x}_{1:i-1}$ to

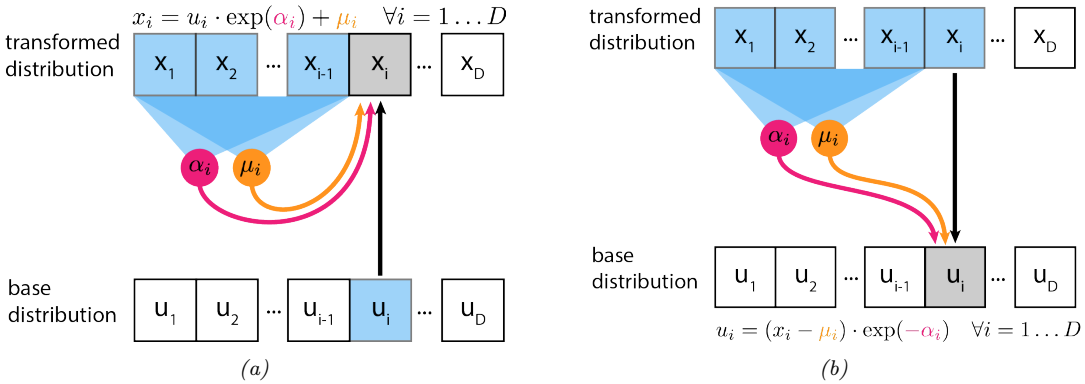


Figure 23.4: (a) Affine autoregressive flow with one layer. In this figure, $\boldsymbol{u}$ is the input to the flow (sample from the base distribution) and $\boldsymbol{x}$ is its output (sample from the transformed distribution). (b) Inverse of the above. From [Jan18]. Used with kind permission of Eric Jang.

compute x_i; therefore, computing the components of $\boldsymbol{x}$ must be done sequentially, by first computing x_1, then using it to compute x_2, then using x_1 and x_2 to compute x_3, and so on. On the other hand, computing the inverse can be done in parallel for each u_i, since $\boldsymbol{u}$ does not appear on the right-hand side of Equation (23.20). Hence, in practice it is often faster to compute $\boldsymbol{f}^{-1}$ than to compute $\boldsymbol{f}$, assuming h and h^{-1} have similar computational cost.

23.2.4.1 Affine autoregressive flows

For a concrete example, we can take h to be an affine scalar bijection (Section 23.2.2.1) parameterized by a log scale α and a bias μ. Such autoregressive flows are known as **affine autoregressive flows**. The parameters of the i'th component, α_i and μ_i, are functions of $\boldsymbol{x}_{1:i-1}$, so $\boldsymbol{f}$ takes the following form:

$$x_i = u_i \exp(\alpha_i(\boldsymbol{x}_{1:i-1})) + \mu_i(\boldsymbol{x}_{1:i-1}). \tag{23.22}$$

This is illustrated in Figure 23.4(a). We can invert this by

$$u_i = (x_i - \mu_i(\boldsymbol{x}_{1:i-1})) \exp(-\alpha_i(\boldsymbol{x}_{1:i-1})). \tag{23.23}$$

This is illustrated in Figure 23.4(b). Finally, we can calculate the log absolute Jacobian determinant by

$$\log |\det \mathbf{J}(\boldsymbol{f})| = \log \left| \prod_{i=1}^{D} \exp(\alpha_i(\boldsymbol{x}_{1:i-1})) \right| = \sum_{i=1}^{D} \alpha_i(\boldsymbol{x}_{1:i-1}). \tag{23.24}$$

Let us look at an example of an affine autoregressive flow on a 2d density estimation problem. Consider an affine autoregressive flow $\boldsymbol{x} = (x_1, x_2) = \boldsymbol{f}(\boldsymbol{u})$, where $\boldsymbol{u} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and $\boldsymbol{f}$ is a single autoregressive bijection. Since x_1 is an affine transformation of $u_1 \sim \mathcal{N}(0, 1)$, it is Gaussian with mean μ_1 and standard deviation $\sigma_1 = \exp \alpha_1$. Similarly, if we consider x_1 fixed, x_2 is an affine

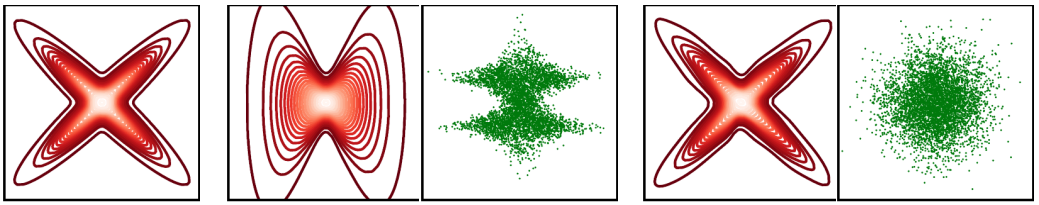


Figure 23.5: Density estimation with affine autoregressive flows, using a Gaussian base distribution. (a) True density. (b) Estimated density using a single autoregressive layer with ordering (x_1, x_2). On the left (contour plot) we show $p(\boldsymbol{x})$. On the right (green dots) we show samples of $\boldsymbol{u} = \boldsymbol{f}^{-1}(\boldsymbol{x})$, where $\boldsymbol{x}$ is sampled from the true density. (c) Same as (b), but using 5 autoregressive layers and reversing the variable ordering after each layer. Adapted from Figure 1 of [PPM17]. Used with kind permission of Iain Murray.

transformation of $u_2 \sim \mathcal{N}(0,1)$, so it is *conditionally* Gaussian with mean $\mu_2(x_1)$ and standard deviation $\sigma_2(x_1) = \exp \alpha_2(x_1)$. Thus, a single affine autoregressive bijection will always produce a distribution with Gaussian conditionals, that is, a distribution of the following form:

$$p(x_1, x_2) = p(x_1)\, p(x_2|x_1) = \mathcal{N}(x_1|\mu_1, \sigma_1^2)\, \mathcal{N}(x_2|\mu_2(x_1), \sigma_2(x_1)^2) \tag{23.25}$$

This result generalizes to an arbitrary number of dimensions D.

A single affine bijection is not very powerful, regardless of how flexible the functions $\alpha_2(x_1)$ and $\mu_2(x_1)$ are. For example, suppose we want to fit the cross-shaped density shown in Figure 23.5(a) with such a flow. The resulting maximum-likelihood fit is shown in Figure 23.5(b). The red contours show the predictive distribution, $\hat{p}(\boldsymbol{x})$, which clearly fails to capture the true distribution. The green dots show transformed versions of the data samples, $p(\boldsymbol{u})$; we see that this is far from the Gaussian base distribution.

Fortunately, we can obtain a better fit by composing multiple autoregressive bijections (layers), and reversing the order of the variables after each layer. For example, Figure 23.5(c) shows the results of an affine autoregressive flow with 5 layers applied to the same problem. The red contours show that we have matched the empirical distribution, and the green dots show we have matched the Gaussian base distribution.

Note that another way to obtain a better fit is to replace the affine bijection h with a more flexible one, such as a monotonic MLP (Section 23.2.2.3) or a monotonic spline (Section 23.2.2.5).

23.2.4.2 Masked autoregressive flows

As we have seen, the conditioners Θ_i can be arbitrary non-linear functions. The most straightforward way to parameterize them is separately for each i, for example by using D separate neural networks. However, this can be parameter-inefficient for large D.

In practice, we often share parameters between conditioners by combining them into a single model Θ that takes in $\boldsymbol{x}$ and outputs $(\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_D)$. For the bijection to remain autoregressive, we must constrain Θ so that $\boldsymbol{\theta}_i$ depends only on $\boldsymbol{x}_{1:i-1}$ and not on $\boldsymbol{x}_{i:D}$. One way to achieve this is to start with an arbitrary neural network (an MLP, a CNN, a ResNet, etc.), and drop connections (for example, by zeroing out weights) until $\boldsymbol{\theta}_i$ is only a function of $\boldsymbol{x}_{1:i-1}$.

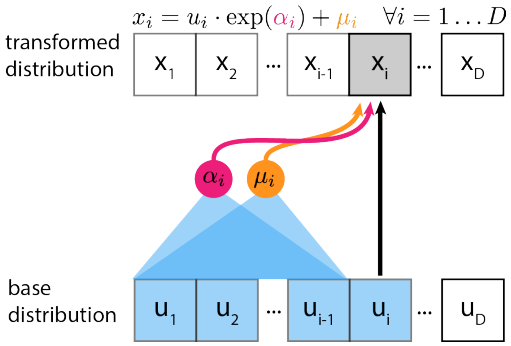


Figure 23.6: Inverse autoregressive flow that uses affine scalar bijections. In this figure, $\boldsymbol{u}$ is the input to the flow (sample from the base distribution) and $\boldsymbol{x}$ is its output (sample from the transformed distribution) From [Jan18]. Used with kind permission of Eric Jang.

An example of this approach is the **masked autoregressive flow** (**MAF**) model of [PPM17]. This model is an affine autoregressive flow combined with permutation layers, as we described in Section 23.2.4.1. MAF implements the combined conditioner Θ as follows: it starts with an MLP, and then multiplies (elementwise) the weight matrix of each layer with a binary mask of the same size (different masks are used for different layers). The masks are constructed using the method of [Ger+15]. This ensures that all computational paths from x_j to $\boldsymbol{\theta}_i$ are zeroed out whenever $j \geq i$, effectively making $\boldsymbol{\theta}_i$ only a function of $\boldsymbol{x}_{1:i-1}$. Still, evaluating the masked conditioner Θ has the same computational cost as evaluating the original (unmasked) MLP.

The key advantage of MAF (and of related models) is that, given $\boldsymbol{x}$, all parameters $(\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_D)$ can be computed efficiently with one neural network evaluation, so the computation of the inverse $\boldsymbol{f}^{-1}$ is fast. Thus, we can efficiently evaluate the probability density of the flow model for arbitrary datapoints. However, in order to compute $\boldsymbol{f}$, the conditioner Θ must be called a total of D times, since not all entries of $\boldsymbol{x}$ are available to start with. Thus, generating new samples from the flow is D times more expensive than evaluating its probability density function. This makes MAF suitable for density estimation, but less so for data generation.

23.2.4.3 Inverse autoregressive flows

As we have seen, the parameters $\boldsymbol{\theta}_i$ that yield the i'th output x_i are functions of the previous outputs $\boldsymbol{x}_{1:i-1}$. This ensures that the Jacobian $\mathbf{J}(\boldsymbol{f})$ is triangular, and so its determinant is efficient to compute.

However, there is another possibility: we can make $\boldsymbol{\theta}_i$ a function of the previous *inputs* instead, that is, a function of $\boldsymbol{u}_{1:i-1}$. This leads to the following bijection, which is known as **inverse autoregressive**:

$$x_i = h(u_i; \Theta_i(\boldsymbol{u}_{1:i-1})), \quad i = 1, \ldots, D. \tag{23.26}$$

Like its autoregressive counterpart, this bijection has a triangular Jacobian whose determinant is also given by $\det \mathbf{J}(\boldsymbol{f}) = \prod_{i=1}^{D} \frac{dh}{du_i}$. Figure 23.6 illustrates an inverse autoregressive flow, for the case where h is affine.

To see why this bijection is called "inverse autoregressive", compare Equation (23.26) with Equation (23.20). The two formulas differ only notationally: we can get from one to the other by swapping $\boldsymbol{u}$ with $\boldsymbol{x}$ and h with h^{-1}. In other words, the inverse autoregressive bijection corresponds to a direct parameterization of the inverse of an autoregressive bijection.

Since inverse autoregressive bijections swap the forwards and inverse directions of their autoregressive counterparts, they also swap their computational properties. This means that the forward direction $\boldsymbol{f}$ of an inverse autoregressive flow is inherently parallel and therefore fast, whereas its inverse direction $\boldsymbol{f}^{-1}$ is inherently sequential and therefore slow.

An example of an inverse autoregressive flow is their namesake **IAF** model of [Kin+16]. IAF uses affine scalar bijections, masked conditioners, and permutation layers, so it is precisely the inverse of the MAF model described in Section 23.2.4.2. Using IAF, we can generate $\boldsymbol{u}$ in parallel from the base distribution (using, for example, a diagonal Gaussian), and then sample each element of $\boldsymbol{x}$ in parallel. However, evaluating $p(\boldsymbol{x})$ for an arbitrary datapoint $\boldsymbol{x}$ is slow, because we have to evaluate each element of $\boldsymbol{u}$ sequentially. Fortunately, evaluating the likelihood of samples generated from IAF (as opposed to externally provided samples) incurs no additional cost, since in this case the u_i terms will already have been computed.

Although not so suitable for density estimation or maximum-likelihood training, IAFs are well-suited for parameterizing variational posteriors in variational inference. This is because in order to estimate the variational lower bound (ELBO), we only need samples from the variational posterior and their associated probability densities, both of which are efficient to obtain. See Section 23.1.2.2 for details.

Another useful application of IAFs is training them to mimic models whose probability density is fast to evaluate but which are slow to sample from. A notable example is the **parallel wavenet** model of [Oor+18]. This model is an IAF p_s that it trained to mimic a pretrained wavenet model p_t by minimizing the KL divergence $D_{\mathbb{KL}}\left(p_s \parallel p_t\right)$. This KL can be easily estimated by first sampling from p_s and then evaluating $\log p_s$ and $\log p_t$ at those samples, operations which are all efficient for these models. After training, we obtain an IAF that can generate audio of similar quality as the original wavenet, but can do so much faster.

23.2.4.4 Connection with autoregressive models

Autoregressive flows can be thought of as generalizing autoregressive models of continuous random variables, discussed in Section 22.1. Specifically, any continuous autoregressive model can be reparameterized as a one-layer autoregressive flow, as we describe below.

Consider a general autoregressive model over a continuous random variable $\boldsymbol{x} = (x_1, \ldots, x_D) \in \mathbb{R}^D$ written as

$$p(\boldsymbol{x}) = \prod_{i=1}^{D} p_i(x_i|\boldsymbol{\theta}_i) \quad \text{where} \quad \boldsymbol{\theta}_i = \Theta_i(\boldsymbol{x}_{1:i-1}). \tag{23.27}$$

In the above expression, $p_i(x_i|\boldsymbol{\theta}_i)$ is the i'th conditional distribution of the autoregressive model, whose parameters $\boldsymbol{\theta}_i$ are arbitrary functions of the previous variables $\boldsymbol{x}_{1:i-1}$. For example, $p_i(x_i|\boldsymbol{\theta}_i)$ can be a mixture of one-dimensional Gaussian distributions, with $\boldsymbol{\theta}_i$ representing the collection of its means, variances, and mixing coefficients.

Now consider sampling a vector $\boldsymbol{x}$ from the autoregressive model, which can be done by sampling

one element at a time as follows:

$$x_i \sim p_i(x_i|\Theta_i(\boldsymbol{x}_{1:i-1})) \quad \text{for } i = 1, \ldots, D. \tag{23.28}$$

Each conditional can be sampled from using inverse transform sampling (Section 11.3.1). Let $U(0,1)$ be the uniform distribution on the interval $[0,1]$, and let $\text{CDF}_i(x_i|\boldsymbol{\theta}_i)$ be the cumulative distribution function of the i'th conditional. Sampling can be written as:

$$x_i = \text{CDF}_i^{-1}(u_i|\Theta_i(\boldsymbol{x}_{1:i-1})) \quad \text{where} \quad u_i \sim U(0,1). \tag{23.29}$$

Comparing the above expression with the definition of an autoregressive bijection in Equation (23.19), we see that the autoregressive model has been expressed as a one-layer autoregressive flow whose base distribution is uniform on $[0,1]^D$ and whose scalar bijections correspond to the inverse conditional cdf's. Viewing autoregressive models as flows this way has an important advantage, namely that it allows us to increase the flexibility of an autoregressive model by composing multiple instances of it in a flow, without sacrificing the overall tractability.

23.2.5 Residual flows

A residual network is a composition of **residual connections**, which are functions of the form $\boldsymbol{f}(\boldsymbol{u}) = \boldsymbol{u} + \mathbf{F}(\boldsymbol{u})$. The function $\mathbf{F} : \mathbb{R}^D \to \mathbb{R}^D$ is called the **residual block**, and it computes the difference between the output and the input, $\boldsymbol{f}(\boldsymbol{u}) - \boldsymbol{u}$.

Under certain conditions on $\mathbf{F}$, the residual connection $\boldsymbol{f}$ becomes invertible. We will refer to flows composed of invertible residual connections as **residual flows**. In the following, we describe two ways the residual block $\mathbf{F}$ can be constrained so that the residual connection $\boldsymbol{f}$ is invertible.

23.2.5.1 Contractive residual blocks

One way to ensure the residual connection is invertible is to choose the residual block to be a contraction. A contraction is a function $\mathbf{F}$ whose Lipschitz constant is less than 1; that is, there exists $0 \leq L < 1$ such that for all $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ we have:

$$\|\mathbf{F}(\boldsymbol{u}_1) - \mathbf{F}(\boldsymbol{u}_2)\| \leq L\|\boldsymbol{u}_1 - \boldsymbol{u}_2\|. \tag{23.30}$$

The invertibility of $\boldsymbol{f}(\boldsymbol{u}) = \boldsymbol{u} + \mathbf{F}(\boldsymbol{u})$ can be shown as follows. Consider the mapping $\boldsymbol{g}(\boldsymbol{u}) = \boldsymbol{x} - \mathbf{F}(\boldsymbol{u})$. Because $\mathbf{F}$ is a contraction, $\boldsymbol{g}$ is also a contraction. So, by Banach's fixed-point theorem, $\boldsymbol{g}$ has a unique fixed point $\boldsymbol{u}_*$. Hence we have

$$\boldsymbol{u}_* = \boldsymbol{x} - \mathbf{F}(\boldsymbol{u}_*) \tag{23.31}$$

$$\Rightarrow \quad \boldsymbol{u}_* + \mathbf{F}(\boldsymbol{u}_*) = \boldsymbol{x} \tag{23.32}$$

$$\Rightarrow \quad \boldsymbol{f}(\boldsymbol{u}_*) = \boldsymbol{x}. \tag{23.33}$$

Because $\boldsymbol{u}_*$ is unique, it follows that $\boldsymbol{u}_* = \boldsymbol{f}^{-1}(\boldsymbol{x})$.

An example of a residual flow with contractive residual blocks is the **iResNet** model of [Beh+19]. The residual blocks of iResNet are convolutional neural networks, that is, compositions of convolutional layers with non-linear activation functions. Because the Lipschitz constant of a composition is less or equal to the product of the Lipschitz constants of the individual functions, it is enough to ensure the

convolutions are contractive, and to use increasing activation functions with slope less or equal to 1. The iResNet model ensures the convolutions are contractive by applying spectral normalization to their weights [Miy+18a].

In general, there is no analytical expression for the inverse $\boldsymbol{f}^{-1}$. However, we can approximate $\boldsymbol{f}^{-1}(\boldsymbol{x})$ using the following iterative procedure:

$$\boldsymbol{u}_n = \boldsymbol{g}(\boldsymbol{u}_{n-1}) = \boldsymbol{x} - \mathbf{F}(\boldsymbol{u}_{n-1}). \tag{23.34}$$

Banach's fixed-point theorem guarantees that the sequence $\boldsymbol{u}_0, \boldsymbol{u}_1, \boldsymbol{u}_2, \ldots$ will converge to $\boldsymbol{u}_* = \boldsymbol{f}^{-1}(\boldsymbol{x})$ for any choice of $\boldsymbol{u}_0$, and it will do so at a rate of $O(L^n)$, where L is the Lipschitz constant of $\boldsymbol{g}$ (which is the same as the Lipschitz constant of $\mathbf{F}$). In practice, it is convenient to choose $\boldsymbol{u}_0 = \boldsymbol{x}$.

In addition, there is no analytical expression for the Jacobian determinant, whose exact computation costs $O(D^3)$. However, there is a computationally efficient stochastic estimator of the log Jacobian determinant. The idea is to express the log Jacobian determinant as a power series. Using the fact that $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{x} + \mathbf{F}(\boldsymbol{x})$, we have

$$\log|\det \mathbf{J}(\boldsymbol{f})| = \log|\det(\mathbf{I} + \mathbf{J}(\mathbf{F}))| = \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k} \mathrm{tr}\left[\mathbf{J}(\mathbf{F})^k\right]. \tag{23.35}$$

This power series converges when the matrix norm of $\mathbf{J}(\mathbf{F})$ is less than 1, which here is guaranteed exactly because $\mathbf{F}$ is a contraction. The trace of $\mathbf{J}(\mathbf{F})^k$ can be efficiently approximated using Jacobian-vector products via the **Hutchinson trace estimator** [Ski89; Hut89; Mey+21]:

$$\mathrm{tr}\left[\mathbf{J}(\mathbf{F})^k\right] \approx \boldsymbol{v}^\top \mathbf{J}(\mathbf{F})^k \boldsymbol{v}, \tag{23.36}$$

where $\boldsymbol{v}$ is a sample from a distribution with zero mean and unit covariance, such as $\mathcal{N}(\mathbf{0}, \mathbf{I})$. Finally, the infinite series can be approximated by a finite one either by truncation [Beh+19], which unfortunately yields a biased estimator, or by employing the **Russian-roulette estimator** [Che+19], which is unbiased.

23.2.5.2 Residual blocks with low-rank Jacobian

There is an efficient way of computing the determinant of a matrix which is a low-rank perturbation of an identity matrix. Suppose $\mathbf{A}$ and $\mathbf{B}$ are matrices, where $\mathbf{A}$ is $D \times M$ and $\mathbf{B}$ is $M \times D$. The following formula is known as the **Weinstein-Aronszajn identity**[2], and is a special case of the more general **matrix determinant lemma**:

$$\det(\mathbf{I}_D + \mathbf{AB}) = \det(\mathbf{I}_M + \mathbf{BA}). \tag{23.37}$$

We write $\mathbf{I}_D$ and $\mathbf{I}_M$ for the $D \times D$ and $M \times M$ identity matrices respectively. The significance of this formula is that it turns a $D \times D$ determinant that costs $O(D^3)$ into an $M \times M$ determinant that costs $O(M^3)$. If M is smaller than D, this saves computation.

With some restrictions on the residual block $\mathbf{F} : \mathbb{R}^D \to \mathbb{R}^D$, we can apply this formula to compute the determinant of a residual connection efficiently. The trick is to create a bottleneck inside $\mathbf{F}$. We do that by defining $\mathbf{F} = \mathbf{F}_2 \circ \mathbf{F}_1$, where $\mathbf{F}_1 : \mathbb{R}^D \to \mathbb{R}^M$, $\mathbf{F}_2 : \mathbb{R}^M \to \mathbb{R}^D$ and $M \ll D$. The chain

2. See https://en.wikipedia.org/wiki/Weinstein-Aronszajn_identity.

rule gives $\mathbf{J}(\mathbf{F}) = \mathbf{J}(\mathbf{F}_2)\mathbf{J}(\mathbf{F}_1)$, where $\mathbf{J}(\mathbf{F}_2)$ is $D \times M$ and $\mathbf{J}(\mathbf{F}_1)$ is $M \times D$. Now we can apply our determinant formula as follows:

$$\det \mathbf{J}(\boldsymbol{f}) = \det(\mathbf{I}_D + \mathbf{J}(\mathbf{F})) = \det(\mathbf{I}_D + \mathbf{J}(\mathbf{F}_2)\mathbf{J}(\mathbf{F}_1)) = \det(\mathbf{I}_M + \mathbf{J}(\mathbf{F}_1)\mathbf{J}(\mathbf{F}_2)). \tag{23.38}$$

Since the final determinant costs $O(M^3)$, we can make the Jacobian determinant efficient by reducing M, that is, by narrowing the bottleneck.

An example of the above is the **planar flow** of [RM15]. In this model, each residual block is an MLP with one hidden layer and one hidden unit. That is,

$$\boldsymbol{f}(\boldsymbol{u}) = \boldsymbol{u} + \boldsymbol{v}\sigma(\boldsymbol{w}^\top \boldsymbol{u} + b), \tag{23.39}$$

where $\boldsymbol{v} \in \mathbb{R}^D$, $\boldsymbol{w} \in \mathbb{R}^D$ and $b \in \mathbb{R}$ are the parameters, and σ is the activation function. The residual block is the composition of $\mathbf{F}_1(\boldsymbol{u}) = \boldsymbol{w}^\top \boldsymbol{u} + b$ and $\mathbf{F}_2(z) = \boldsymbol{v}\sigma(z)$, so $M = 1$. Their Jacobians are $\mathbf{J}(\mathbf{F}_1)(\boldsymbol{u}) = \boldsymbol{w}^\top$ and $\mathbf{J}(\mathbf{F}_2)(z) = \boldsymbol{v}\sigma'(z)$. Substituting these in the formula for the Jacobian determinant we obtain:

$$\det \mathbf{J}(\boldsymbol{f})(\boldsymbol{u}) = 1 + \boldsymbol{w}^\top \boldsymbol{v}\sigma'(\boldsymbol{w}^\top \boldsymbol{u} + b), \tag{23.40}$$

which can be computed efficiently in $O(D)$. Other examples include the **circular flow** of [RM15] and the **Sylvester flow** of [Ber+18].

This technique gives an efficient way of computing determinants of residual connections with bottlenecks, but in general there is no guarantee that such functions are invertible. This means that invertibility must be satisfied on a case-by-case basis. For example, the planar flow is invertible when σ is the hyperbolic tangent and $\boldsymbol{w}^\top \boldsymbol{v} > -1$, but otherwise it may not be.

23.2.6 Continuous-time flows

So far we have discussed flows that consist of a sequence of bijections $\boldsymbol{f}_1, \ldots, \boldsymbol{f}_N$. Starting from some input $\boldsymbol{x}_0 = \boldsymbol{u}$, this creates a sequence of outputs $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$ where $\boldsymbol{x}_n = \boldsymbol{f}_n(\boldsymbol{x}_{n-1})$. However, we can also have flows where the input is transformed into the final output in a continuous way. That is, we start from $\boldsymbol{x}_0 = \boldsymbol{x}(0)$, create a continuously-indexed sequence $\boldsymbol{x}(t)$ for $t \in [0, T]$ with some fixed T, and take $\boldsymbol{x}(T)$ to be the final output. Thinking of t as analogous to time, we refer to these as **continuous-time flows**.

The sequence $\boldsymbol{x}(t)$ is defined as the solution to a first-order ordinary differential equation (ODE) of the form:

$$\frac{d\boldsymbol{x}}{dt}(t) = \mathbf{F}(\boldsymbol{x}(t), t). \tag{23.41}$$

The function $\mathbf{F} : \mathbb{R}^D \times [0, T] \to \mathbb{R}^D$ is a time-dependent vector field that parameterizes the ODE. If we think of $\boldsymbol{x}(t)$ as the position of a particle in D dimensions, the vector $\mathbf{F}(\boldsymbol{x}(t), t)$ determines the particle's velocity at time t.

The flow (for time T) is a function $\boldsymbol{f} : \mathbb{R}^D \to \mathbb{R}^D$ that takes in an input $\boldsymbol{x}_0$, solves the ODE with initial condition $\boldsymbol{x}(0) = \boldsymbol{x}_0$, and returns $\boldsymbol{x}(T)$. The function $\boldsymbol{f}$ is a well-defined bijection if the solution to the ODE exists for all $t \in [0, T]$ and is unique. These conditions are not generally satisfied for arbitrary $\mathbf{F}$, but they are if $\mathbf{F}(\cdot, t)$ is Lipschitz continuous with a Lipschitz constant that does not

depend on t. That is, $\boldsymbol{f}$ is a well-defined bijection if there exists a constant L such that for all $\boldsymbol{x}_1$, $\boldsymbol{x}_2$ and $t \in [0, T]$ we have:

$$\|\mathbf{F}(\boldsymbol{x}_1, t) - \mathbf{F}(\boldsymbol{x}_2, t)\| \leq L\|\boldsymbol{x}_1 - \boldsymbol{x}_2\|. \tag{23.42}$$

This result is a consequence of the **Picard-Lindelöf theorem** for ODEs.[3] In practice, we can parameterize $\mathbf{F}$ using any choice of model, provided the Lipschitz condition is met.

Usually the ODE cannot be solved analytically, but we can solve it approximately by discretizing it. A simple example is **Euler's method**, which corresponds to the following discretization for some small step size $\epsilon > 0$:

$$\boldsymbol{x}(t + \epsilon) = \boldsymbol{x}(t) + \epsilon\mathbf{F}(\boldsymbol{x}(t), t). \tag{23.43}$$

This is equivalent to a residual connection with residual block $\epsilon\mathbf{F}(\cdot, t)$, so the ODE solver can be thought of as a deep residual network with $O(T/\epsilon)$ layers. A smaller step size leads to a more accurate solution, but also to more computation. There are several other solution methods varying in accuracy and sophistication, such as those in the broader Runge-Kutta family, some of which use adaptive step sizes.

The inverse of $\boldsymbol{f}$ can be easily computed by solving the ODE in reverse. That is, to compute $\boldsymbol{f}^{-1}(\boldsymbol{x}_T)$ we solve the ODE with initial condition $\boldsymbol{x}(T) = \boldsymbol{x}_T$, and return $\boldsymbol{x}(0)$. Unlike some other flows (such as autoregressive flows) which are more expensive to compute in one direction than in the other, continuous-time flows require the same amount of computation in either direction.

In general, there is no analytical expression for the Jacobian determinant of $\boldsymbol{f}$. However, we can express it as the solution to a separate ODE, which we can then solve numerically. First, we define $\boldsymbol{f}_t : \mathbb{R}^D \to \mathbb{R}^D$ to be the flow for time t, that is, the function that takes $\boldsymbol{x}_0$, solves the ODE with initial condition $\boldsymbol{x}(0) = \boldsymbol{x}_0$ and returns $\boldsymbol{x}(t)$. Clearly, $\boldsymbol{f}_0$ is the identity function and $\boldsymbol{f}_T = \boldsymbol{f}$. Let us define $L(t) = \log|\det \mathbf{J}(\boldsymbol{f}_t)(\boldsymbol{x}_0)|$. Because $\boldsymbol{f}_0$ is the identity function, $L(0) = 0$, and because $\boldsymbol{f}_T = \boldsymbol{f}$, $L(T)$ gives the Jacobian determinant of $\boldsymbol{f}$ that we are interested in. It can be shown that L satisfies the following ODE:

$$\frac{dL}{dt}(t) = \operatorname{tr}\big[\mathbf{J}(\mathbf{F}(\cdot, t))(\boldsymbol{x}(t))\big]. \tag{23.44}$$

That is, the rate of change of L at time t is equal to the Jacobian trace of $\mathbf{F}(\cdot, t)$ evaluated at $\boldsymbol{x}(t)$. So we can compute $L(T)$ by solving the above ODE with initial condition $L(0) = 0$. Moreover, we can compute $\boldsymbol{x}(T)$ and $L(T)$ simultaneously, by combining their two ODEs into a single ODE operating on the extended space $(\boldsymbol{x}, L)$.

An example of a continuous-time flow is the **neural ODE** model of [Che+18c], which uses a neural network to parameterize $\mathbf{F}$. To avoid backpropagating gradients through the ODE solver, which can be computationally demanding, they use the **adjoint sensitivity method** to express the time evolution of the gradient with respect to $\boldsymbol{x}(t)$ as a separate ODE. Solving this ODE gives the required gradients, and can be thought of as the continuous-time analog of backpropagation.

Another example is the **FFJORD** model of [Gra+19]. This is similar to the neural ODE model, except that it uses the Hutchinson trace estimator to approximate the Jacobian trace of $\mathbf{F}(\cdot, t)$. This usage of the Hutchinson trace estimator is analogous to that in contractive residual flows (Section 23.2.5.1), and it speeds up computation in exchange for a stochastic (but unbiased) estimate.

See also Section 25.4.4, where we discuss continuous time diffusion models.

3. See https://en.wikipedia.org/wiki/Picard-Lindel%C3%B6f_theorem

23.3 Applications

In this section, we highlight some applications of flows for canonical probabilistic machine learning tasks.

23.3.1 Density estimation

Flow models allow exact density computation and can be used to fit multi-modal densities to observed data. (see Figure 23.3 for an example). An early example is Gaussianization [CG00] who applied this idea to fit low-dimensional densities. Tabak and Vanden-Eijnden [TVE10] and Tabak and Turner [TT13] introduced the modern idea of flows (including the term 'normalizing flows'), describing a flow as a composition of simpler maps. Deep density models [RA13] was one of the first to use neural networks for flows to parameterize high-dimensional densities. There has been a rich line of follow-up work including **NICE** [DKB15] and **Real NVP** [DSDB17]. (NVP stands for "non-volume-preserving", which refers to the fact that the Jacobian of the transform is not unity.) Masked autoregressive flows (Section 23.2.4.2) further improved performance on unconditional and conditional density estimation tasks.

Flows can be used for *hybrid models* which model the joint density of inputs and targets $p(\boldsymbol{x}, y)$, as opposed to discriminative classification models which just model the conditional $p(y|\boldsymbol{x})$ and density models which just model the marginal $p(\boldsymbol{x})$. Nalisnick et al. [Nal+19b] proposed a flow-based hybrid model using invertible mappings for representation learning and showed that the joint density $p(\boldsymbol{x}, y)$ can be computed efficiently, which can be useful for downstream tasks such as anomaly detection, semi-supervised learning and selective classification. Flow-based hybrid models are memory-efficient since most of the parameters are in the invertible representation which are shared between the discriminative and generative models; furthermore, the density $p(\boldsymbol{x}, y)$ can be computed in a single forwards pass leading to computational savings. Residual flows [Che+19] use invertible residual mappings [Beh+19] for hybrid modeling which further improves performance. Flows have also been used to fit densities to embeddings [Zha+20b; CZG20] for anomaly detection tasks.

23.3.2 Generative modeling

Another task is generation, which involves generating novel samples from a fitted model $p^*(\boldsymbol{x})$. Generation is a popular downstream task for normalizing flows, which have been applied for different data modalities including images, video, audio, text, and structured objects such as graphs and point clouds. Images are arguably the most popular modality for deep generative models: GLOW [KD18b] was one of the first flow-based models to generate compelling high-dimensional images, and has been extended to video to produce RGB frames [Kum+19b]; residual flows [Che+19] have also been shown to produce sharp images.

Oord et al. [Oor+18] used flows for audio synthesis by distilling WaveNet into an IAF (Section 23.2.4.3), which enables faster sampling than WaveNet. Other flow models for audio include WaveFLOW [PVC19] and FlowWaveNet [Kim+19], which directly speed up WaveNet using coupling layers.

Flows have been also used for text. Tran et al. [Tra+19] define a discrete flow over a vocabulary for language-modeling tasks. Another popular approach is to define a latent variable model with discrete observation space but a continuous latent space. For example, Ziegler and Rush [ZR19a] use

normalizing flows in latent space for language modeling.

23.3.3 Inference

Normalizing flows have been used for probabilistic inference. Rezende and Mohamed [RM15] popularized normalizing flows in machine learning, and showed how they can be used for modeling variational posterior distributions in latent variable models. Various extensions such as Householder flows [TW16], inverse autoregressive flows [Kin+16], multiplicative normalizing flows [LW17], and Sylvester flows [Ber+18] have been proposed for modeling the variational posterior for latent variable models, as well as posteriors for Bayesian neural networks.

Flows have been used as complex proposal distributions for importance sampling; examples include neural importance sampling [Mül+19b] and Boltzmann generators [Noé+19]. Hoffman et al. [Hof+19] used flows to improve the performance of Hamiltonian Monte Carlo (Section 12.5) by defining bijective transformations to transform random variables to simpler distributions and performing HMC in that space instead.

Finally, flows can be used in the context of simulation-based inference, where the likelihood function of the parameters is not available, but simulating data from the model is possible. The main idea is to train a flow on data simulated from the model in order to approximate the posterior distribution or the likelihood function. The flow model can also be used to guide simulations in order to make inference more efficient [PSM19; GNM19]. This approach has been used for inference of simulation models in cosmology [Als+19] and computational neuroscience [Gon+20].

24 Energy-based models

This chapter is co-authored with Yang Song and Durk Kingma.

24.1 Introduction

We have now seen several ways of defining deep generative models, including VAEs (Chapter 21), autoregressive models (Chapter 22), and normalizing flows (Chapter 23). All of the above models can be formulated in terms of directed graphical models (Chapter 4), where we generate the data one step at a time, using locally normalized distributions. In some cases, it is easier to specify a distribution in terms of a set of constraints that valid samples must satisfy, rather than a generative process. This can be done using an undirected graphical model (Chapter 4).

Energy-based models or **EBM** can be written as a Gibbs distribution as follows:

$$p_{\boldsymbol{\theta}}(\mathbf{x}) = \frac{\exp(-\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x}))}{Z_{\boldsymbol{\theta}}} \tag{24.1}$$

where $\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x}) \geq 0$ is known as the **energy function** with parameters $\boldsymbol{\theta}$, and $Z_{\boldsymbol{\theta}}$ is the **partition function**:

$$Z_{\boldsymbol{\theta}} = \int \exp(-\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x}))\, \mathrm{d}\mathbf{x} \tag{24.2}$$

This is constant wrt $\mathbf{x}$ but is a function of $\boldsymbol{\theta}$. Since EBMs do not usually make any Markov assumptions (unlike graphical models), evaluating this integral is usually intractable. Consequently we usually need to use approximate methods, such as annealed importance sampling, discussed in Section 11.5.4.1.

The advantage of an EBM over other generative models is that the energy function can be any kind of function that returns a non-negative scalar; it does not need to integrate to 1. This allows one to use a variety of neural network architectures for defining the energy. As such, EBMs have found wide applications in many fields of machine learning, including image generation [Ngi+11; Xie+16; DM19b], discriminative learning [Gra+20b], natural processing [Mik+13; Den+20], density estimation [Wen+19a; Son+19], and reinforcement learning [Haa+17; Haa+18a], to list a few. (More examples can be found at https://github.com/yataobian/awesome-ebm.)

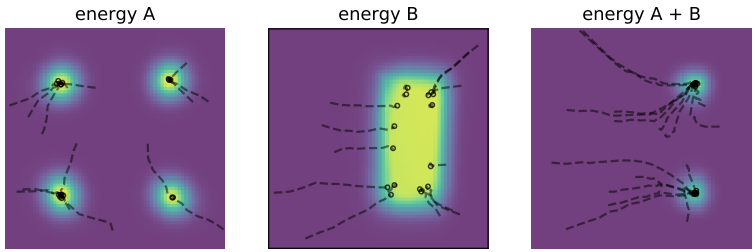


Figure 24.1: Combining two energy functions in 2d by summation, which is equivalent to multiplying the corresponding probability densities. We also illustrate some sampled trajectories towards high probability (low energy) regions. From Figure 14 of [DM19a]. Used with kind permission of Yilun Du.

24.1.1 Example: products of experts (PoE)

As an example of why energy based models are useful, suppose we want to create a generative model of proteins that are thermally stable at room temperature, and which bind to the COVID-19 spike receptor. Suppose $p_1(\boldsymbol{x})$ can generate stable proteins and $p_2(\boldsymbol{x})$ can generate proteins that bind. (For example, both of these models could be autoregressive sequence models, trained on different datasets.) We can view each of these models as "experts" about a particular aspect of the data. On their own, they are not an adequate model of the data that we have (or want to have), but we can then combine them, to represent the **conjunction of features**, by computing a **product of experts** (**PoE**) [Hin02]:

$$p_{12}(\boldsymbol{x}) = \frac{1}{Z_{12}} p_1(\boldsymbol{x}) p_2(\boldsymbol{x}) \tag{24.3}$$

This will assign high probability to proteins that are stable and which bind, and low probability to all others. By contrast, a **mixture of experts** would either generate from p_1 or from p_2, but would not combine features from both.

If the experts are represented as energy based models (EBM), then the PoE model is also an EBM, with an energy given by

$$\mathcal{E}_{12}(\boldsymbol{x}) = \mathcal{E}_1(\boldsymbol{x}) + \mathcal{E}_2(\boldsymbol{x}) \tag{24.4}$$

Intuitively, we can think of each component of energy as a "**soft constraint**" on the data. This idea is illustrated in Figure 24.1.

24.1.2 Computational difficulties

Although the flexibility of EBMs can provide significant modeling advantages, computation of the likelihood and drawing samples from the model are generally intractable. In this chapter, we will discuss a variety of approximate methods to solve these problems.

24.2 Maximum likelihood training

The de facto standard for learning probabilistic models from iid data is maximum likelihood estimation (MLE). Let $p_{\boldsymbol{\theta}}(\mathbf{x})$ be a probabilistic model parameterized by $\boldsymbol{\theta}$, and $p_{\mathcal{D}}(\mathbf{x})$ be the underlying data distribution of a dataset. We can fit $p_{\boldsymbol{\theta}}(\mathbf{x})$ to $p_{\mathcal{D}}(\mathbf{x})$ by maximizing the expected log-likelihood function over the data distribution, defined by

$$\ell(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}\sim p_{\mathcal{D}}(\mathbf{x})}[\log p_{\boldsymbol{\theta}}(\mathbf{x})] \tag{24.5}$$

as a function of $\boldsymbol{\theta}$. Here the expectation can be easily estimated with samples from the dataset. Maximizing likelihood is equivalent to minimizing the KL divergence between $p_{\mathcal{D}}(\mathbf{x})$ and $p_{\boldsymbol{\theta}}(\mathbf{x})$, because

$$\ell(\boldsymbol{\theta}) = -D_{\mathbb{KL}}\left(p_{\mathcal{D}}(\mathbf{x}) \parallel p_{\boldsymbol{\theta}}(\mathbf{x})\right) + \text{const} \tag{24.6}$$

where the constant is equal to $\mathbb{E}_{\mathbf{x}\sim p_{\mathcal{D}}(\mathbf{x})}[\log p_{\mathcal{D}}(\mathbf{x})]$ which does not depend on $\boldsymbol{\theta}$.

We cannot usually compute the likelihood of an EBM because the normalizing constant $Z_{\boldsymbol{\theta}}$ is often intractable. Nevertheless, we can still estimate the gradient of the log-likelihood with MCMC approaches, allowing for likelihood maximization with stochastic gradient ascent [You99]. In particular, the gradient of the log-probability of an EBM decomposes as a sum of two terms:

$$\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\mathbf{x}) = -\nabla_{\boldsymbol{\theta}} \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x}) - \nabla_{\boldsymbol{\theta}} \log Z_{\boldsymbol{\theta}}. \tag{24.7}$$

The first gradient term, $-\nabla_{\boldsymbol{\theta}} \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})$, is straightforward to evaluate with automatic differentiation. The challenge is in approximating the second gradient term, $\nabla_{\boldsymbol{\theta}} \log Z_{\boldsymbol{\theta}}$, which is intractable to compute exactly. This gradient term can be rewritten as the following expectation:

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} \log Z_{\boldsymbol{\theta}} &= \nabla_{\boldsymbol{\theta}} \log \int \exp(-\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})) d\mathbf{x} && (24.8)\\
&\overset{(i)}{=} \left(\int \exp(-\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})) d\mathbf{x}\right)^{-1} \nabla_{\boldsymbol{\theta}} \int \exp(-\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})) d\mathbf{x} && (24.9)\\
&= \left(\int \exp(-\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})) d\mathbf{x}\right)^{-1} \int \nabla_{\boldsymbol{\theta}} \exp(-\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})) d\mathbf{x} && (24.10)\\
&\overset{(ii)}{=} \left(\int \exp(-\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})) d\mathbf{x}\right)^{-1} \int \exp(-\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x}))(-\nabla_{\boldsymbol{\theta}} \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})) d\mathbf{x} && (24.11)\\
&= \int \left(\int \exp(-\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})) d\mathbf{x}\right)^{-1} \exp(-\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x}))(-\nabla_{\boldsymbol{\theta}} \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})) d\mathbf{x} && (24.12)\\
&\overset{(iii)}{=} \int \frac{1}{Z_{\boldsymbol{\theta}}} \exp(-\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x}))(-\nabla_{\boldsymbol{\theta}} \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})) d\mathbf{x} && (24.13)\\
&\overset{(iv)}{=} \int p_{\boldsymbol{\theta}}(\mathbf{x})(-\nabla_{\boldsymbol{\theta}} \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})) d\mathbf{x} && (24.14)\\
&= \mathbb{E}_{\mathbf{x}\sim p_{\boldsymbol{\theta}}(\mathbf{x})}\left[-\nabla_{\boldsymbol{\theta}} \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})\right], && (24.15)
\end{aligned}$$

where steps (i) and (ii) are due to the chain rule of gradients, and (iii) and (iv) are from definitions in Equations (24.1) and (24.2). Thus, we can obtain an unbiased Monte Carlo estimate of the log-likelihood gradient by using

$$\nabla_{\boldsymbol{\theta}} \log Z_{\boldsymbol{\theta}} \simeq -\frac{1}{S} \sum_{s=1}^{S} \nabla_{\boldsymbol{\theta}} \mathcal{E}_{\boldsymbol{\theta}}(\tilde{\mathbf{x}}_s), \tag{24.16}$$

where $\tilde{\mathbf{x}}_s \sim p_{\boldsymbol{\theta}}(\mathbf{x})$, i.e., a random sample from the distribution over $\mathbf{x}$ given by the EBM. Therefore, as long as we can draw random samples from the model, we have access to an unbiased Monte Carlo estimate of the log-likelihood gradient, allowing us to optimize the parameters with stochastic gradient ascent.

Much of the literature has focused on methods for efficient MCMC sampling from EBMs. We discuss some of these methods below.

24.2.1 Gradient-based MCMC methods

Some efficient MCMC methods, such as **Langevin MCMC** (Section 12.5.6) or Hamiltonian Monte Carlo (Section 12.5), make use of the fact that the gradient of the log-probability wrt $\mathbf{x}$ (known as the **score function**) is equal to the (negative) gradient of the energy, and is therefore easy to calculate:

$$\nabla_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\mathbf{x}) = -\nabla_{\mathbf{x}} \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x}) - \underbrace{\nabla_{\mathbf{x}} \log Z_{\boldsymbol{\theta}}}_{=0} = -\nabla_{\mathbf{x}} \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x}). \tag{24.17}$$

For example, when using Langevin MCMC to sample from $p_{\boldsymbol{\theta}}(\mathbf{x})$, we first draw an initial sample $\mathbf{x}^0$ from a simple prior distribution, and then simulate an overdamped Langevin diffusion process for K steps with step size $\epsilon > 0$:

$$\mathbf{x}^{k+1} \leftarrow \mathbf{x}^k + \frac{\epsilon^2}{2} \underbrace{\nabla_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\mathbf{x}^k)}_{=-\nabla_{\mathbf{x}} \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})} + \epsilon \mathbf{z}^k, \quad k = 0, 1, \cdots, K-1. \tag{24.18}$$

where $\boldsymbol{z}^k \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ is a Gaussian noise term. We show an example of this process in Figure 25.5d.

When $\epsilon \to 0$ and $K \to \infty$, $\mathbf{x}^K$ is guaranteed to distribute as $p_{\boldsymbol{\theta}}(\mathbf{x})$ under some regularity conditions. In practice we have to use a small finite ϵ, but the discretization error is typically negligible, or can be corrected with a Metropolis-Hastings step (Section 12.2), leading to the Metropolis-adjusted Langevin algorithm (Section 12.5.6).

24.2.2 Contrastive divergence

Running MCMC till convergence to obtain a sample $\mathbf{x} \sim p_{\boldsymbol{\theta}}(\mathbf{x})$ can be computationally expensive. Therefore we typically need approximations to make MCMC-based learning of EBMs practical. One popular method for doing so is **contrastive divergence** (CD) [Hin02]. In CD, one initializes the MCMC chain from the datapoint $\mathbf{x}$, and proceeds to perform MCMC for a fixed number of steps. One can show that T steps of CD minimizies the following objective:

$$\mathrm{CD}_T = D_{\mathbb{KL}}\left(p_0 \parallel p_\infty\right) - D_{\mathbb{KL}}\left(p_T \parallel p_\infty\right) \tag{24.19}$$

where p_T is the distribution over $\mathbf{x}$ after T MCMC updates, and p_0 is the data distribution. Typically we can get good results with a small value of T, sometimes just $T = 1$. We give the details below.

24.2.2.1 Fitting RBMs with CD

CD was initially developed to fit a special kind of latent variable EBM known as a restricted Boltzmann machine (Section 4.3.3.2). This model was specifically designed to support fast block Gibbs sampling, which is required by CD (and can also be exploited by standard MCMC-based learning methods [AHS85].)

For simplicity, we will assume the hidden and visible nodes are binary, and we use 1-step contrastive divergence. As discussed in Supplementary Section 4.3.1, the binary RBM has the following energy function:

$$\mathcal{E}(\boldsymbol{x}, \boldsymbol{z}; \boldsymbol{\theta}) = \sum_{d=1}^{D}\sum_{k=1}^{K} x_d z_k W_{dk} + \sum_{d=1}^{D} x_d b_d + \sum_{k=1}^{K} z_k c_k \tag{24.20}$$

(Henceforth we will drop the unary (bias) terms, which can be emulated by clamping $z_k = 1$ or $x_d = 1$.) This is a loglinear model where we have one binary feature per edge. Thus from Equation (4.135) the gradient of the log-likelihood is given by the clamped expectations minus the unclamped expectations:

$$\frac{\partial \ell}{\partial w_{dk}} = \frac{1}{N}\sum_{n=1}^{N} \mathbb{E}\left[x_d z_k | \boldsymbol{x}_n, \boldsymbol{\theta}\right] - \mathbb{E}\left[x_d z_k | \boldsymbol{\theta}\right] \tag{24.21}$$

We can rewrite the above gradient in matrix-vector form as follows:

$$\nabla_{\boldsymbol{w}}\, \ell = \mathbb{E}_{p_{\mathcal{D}}(\boldsymbol{x}) p(\boldsymbol{z}|\boldsymbol{x}, \boldsymbol{\theta})}\left[\boldsymbol{x}\boldsymbol{z}^T\right] - \mathbb{E}_{p(\boldsymbol{z}, \boldsymbol{x}|\boldsymbol{\theta})}\left[\boldsymbol{x}\boldsymbol{z}^T\right] \tag{24.22}$$

(We can derive a similar expression for the gradient of the bias terms by setting $x_d = 1$ or $z_k = 1$.)

The first term in the expression for the gradient in Equation (24.21), when $\boldsymbol{x}$ is fixed to a data case, is sometimes called the **clamped phase**, and the second term, when $\boldsymbol{x}$ is free, is sometimes called the **unclamped phase**. When the model expectations match the empirical expectations, the two terms cancel out, the gradient becomes zero and learning stops.

We can also make a connection to the principle of **Hebbian learning** in neuroscience. In particular, Hebb's rule says that the strength of connection between two neurons that are simultaneously active should be increased. (This theory is often summarized as "Cells that fire together wire together".[1]) The first term in Equation (24.21) is therefore considered a Hebbian term, and the second term an anti-Hebbian term, due to the sign change.

We can leverage the Markov structure of the bipartite graph to approximate the expectations as follows:

$$\boldsymbol{z}_n \sim p(\boldsymbol{z}|\boldsymbol{x}_n, \boldsymbol{\theta}) \tag{24.23}$$

$$\boldsymbol{x}'_n \sim p(\boldsymbol{x}|\boldsymbol{z}_n, \boldsymbol{\theta}) \tag{24.24}$$

$$\boldsymbol{z}'_n \sim p(\boldsymbol{z}|\boldsymbol{x}'_n, \boldsymbol{\theta}) \tag{24.25}$$

1. See https://en.wikipedia.org/wiki/Hebbian_theory.

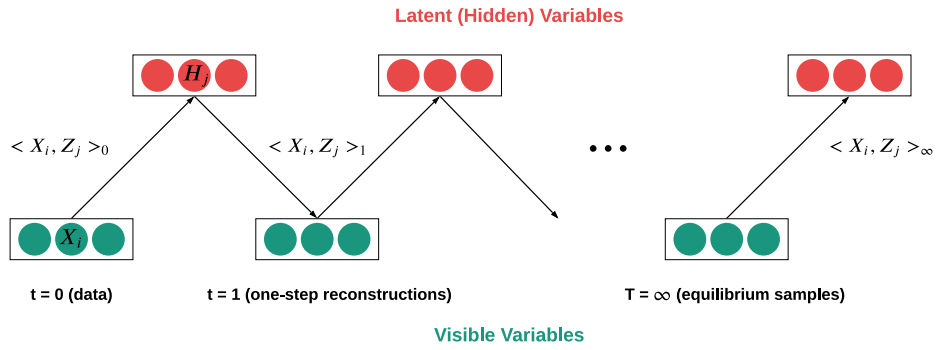


Figure 24.2: Illustration of contrastive divergence sampling for an RBM. The visible nodes are initialized at an example drawn from the dataset. Then we sample a hidden vector, then another visible vector, etc. Eventually (at "infinity") we will be producing samples from the joint distribution $p(\boldsymbol{x}, \boldsymbol{z}|\boldsymbol{\theta})$.

We can think of $\boldsymbol{x}'_n$ as the model's best attempt at reconstructing $\boldsymbol{x}_n$ after being encoded and then decoded by the model. Such samples are sometimes called **fantasy data**. See Figure 24.2 for an illustration. Given these samples, we then make the approximation

$$\mathbb{E}_{p(\cdot|\boldsymbol{\theta})}\left[\boldsymbol{x}\boldsymbol{z}^\mathsf{T}\right] \approx \boldsymbol{x}_n(\boldsymbol{z}'_n)^\mathsf{T} \tag{24.26}$$

In practice, it is common to use $\mathbb{E}\left[\boldsymbol{z}|\boldsymbol{x}'_n\right]$ instead of a sampled value $\boldsymbol{z}'_n$ in the above expression, since this reduces the variance. However, it is not valid to use $\mathbb{E}\left[\boldsymbol{z}|\boldsymbol{x}_n\right]$ instead of sampling $\boldsymbol{z}_n \sim p(\boldsymbol{z}|\boldsymbol{x}_n)$ in Equation (24.23), because then each hidden unit would be able to pass more than 1 bit of information, so it would not act as much of a bottleneck.

The whole procedure is summarized in Algorithm 24.1. For more details, see [Hin10; Swe+10].

Algorithm 24.1: CD-1 training for an RBM with binary hidden and visible units

Initialize weights $\mathbf{W} \in \mathbb{R}^{D\times K}$ randomly
for $t = 1, 2, \ldots$ **do**
  for *each minibatch of size* B **do**
    Set minibatch gradient to zero, $\boldsymbol{g} := \mathbf{0}$
    for *each case* $\boldsymbol{x}_n$ *in the minibatch* **do**
      Compute $\boldsymbol{\mu}_n = \mathbb{E}\left[\boldsymbol{z}|\boldsymbol{x}_n, \mathbf{W}\right]$
      Sample $\boldsymbol{z}_n \sim p(\boldsymbol{z}|\boldsymbol{x}_n, \mathbf{W})$
      Sample $\boldsymbol{x}'_n \sim p(\boldsymbol{x}|\boldsymbol{z}_n, \mathbf{W})$
      Compute $\boldsymbol{\mu}'_n = \mathbb{E}\left[\boldsymbol{z}|\boldsymbol{x}'_n, \mathbf{W}\right]$
      Compute gradient $\nabla_{\mathbf{W}} = (\boldsymbol{x}_n)(\boldsymbol{\mu}_n)^\mathsf{T} - (\boldsymbol{x}'_n)(\boldsymbol{\mu}'_n)^\mathsf{T}$
      Accumulate $\boldsymbol{g} := \boldsymbol{g} + \nabla_{\mathbf{W}}$
    Update parameters $\mathbf{W} := \mathbf{W} + \eta_t \frac{1}{B}\boldsymbol{g}$

24.2.2.2 Persistent CD

One variant of CD that sometimes performs better is **persistent contrastive divergence** (PCD) [Tie08; TH09; You99]. In this approach, a single MCMC chain with a persistent state is employed to sample from the EBM. In PCD, we do not restart the MCMC chain when training on a new datapoint; rather, we carry over the state of the previous MCMC chain and use it to initialize a new MCMC chain for the next training step. See Line 12 for some pseudocode. Hence there are two dynamical processes running at different time scales: the states $\boldsymbol{x}$ change quickly, and the parameters $\boldsymbol{\theta}$ change slowly.

Algorithm 24.2: Persistent MCMC-SGD for fitting an EBM

1 Initialize parameters $\boldsymbol{\theta}$ randomly
2 Initialize chains $\tilde{\boldsymbol{x}}_{1:S}$ randomly
3 Initialize learning rate η
4 **for** $t = 1, 2, \ldots$ **do**
5 **for** $\boldsymbol{x}_b$ *in minibatch of size* B **do**
6 $\boldsymbol{g}_b = \nabla_{\boldsymbol{\theta}} \mathcal{E}_{\boldsymbol{\theta}}(\boldsymbol{x}_b)$
7 **for** *sample* $s = 1 : S$ **do**
8 Sample $\tilde{\boldsymbol{x}}_s \sim \text{MCMC}(\text{target} = p(\cdot|\boldsymbol{\theta}), \text{init} = \tilde{\boldsymbol{x}}_s, \text{nsteps} = N)$
9 $\tilde{\boldsymbol{g}}_s = \nabla_{\boldsymbol{\theta}} \mathcal{E}_{\boldsymbol{\theta}}(\tilde{\boldsymbol{x}}_s)$
10 $\boldsymbol{g}_t = -(\frac{1}{B}\sum_{b=1}^{B} \boldsymbol{g}_b) - (\frac{1}{S}\sum_{s=1}^{S} \tilde{\boldsymbol{g}}_s)$
11 $\boldsymbol{\theta} := \boldsymbol{\theta} + \eta \boldsymbol{g}_t$
12 Decrease step size η

A theoretical justification for this was given in [You89], who showed that we can start the MCMC chain at its previous value, and just take a few steps, because $p(\boldsymbol{x}|\boldsymbol{\theta}_t)$ is likely to be close to $p(\boldsymbol{x}|\boldsymbol{\theta}_{t-1})$, since we only changed the parameters by a small amount in the intervening SGD step.

24.2.2.3 Other methods

PCD can be further improved by keeping multiple historical states of the MCMC chain in a replay buffer and initialize new MCMC chains by randomly sampling from it [DM19b]. Other variants of CD include mean field CD [WH02], and multi-grid CD [Gao+18].

EBMs trained with CD may not capture the data distribution faithfully, since truncated MCMC can lead to biased gradient updates that hurt the learning dynamics [SMB10; FI10; Nij+19]. There are several methods that focus on removing this bias for improved MCMC training. For example, one line of work proposes unbiased estimators of the gradient through coupled MCMC [JOA17; QZW19]; and Du et al. [Du+20] propose to reduce the bias by differentiating through the MCMC sampling algorithm and estimating an entropy correction term.

24.3 Score matching (SM)

If two continuously differentiable real-valued functions $f(\mathbf{x})$ and $g(\mathbf{x})$ have equal first derivatives everywhere, then $f(\mathbf{x}) \equiv g(\mathbf{x}) + \text{constant}$. When $f(\mathbf{x})$ and $g(\mathbf{x})$ are log probability density functions (pdf's) with equal first derivatives, the normalization requirement (Equation (24.1)) implies that $\int \exp(f(\mathbf{x}))\mathrm{d}\mathbf{x} = \int \exp(g(\mathbf{x}))\mathrm{d}\mathbf{x} = 1$, and therefore $f(\mathbf{x}) \equiv g(\mathbf{x})$. As a result, one can learn an EBM by (approximately) matching the first derivatives of its log-pdf to the first derivatives of the log pdf of the data distribution. If they match, then the EBM captures the data distribution exactly. The first-order gradient function of a log pdf wrt its input, $\nabla_{\boldsymbol{x}} \log p_{\boldsymbol{\theta}}(\boldsymbol{x})$, is called the (Stein) **score** function. (This is distinct from the Fisher score, $\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\boldsymbol{x})$.) For training EBMs, it is useful to transform the equivalence of distributions to the equivalence of scores, because the score of an EBM can be easily obtained as follows:

$$\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x}) \triangleq \nabla_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\mathbf{x}) = -\nabla_{\mathbf{x}} \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x}) \tag{24.27}$$

We see that this does not involve the typically intractable normalizing constant $Z_{\boldsymbol{\theta}}$.

Let $p_{\mathcal{D}}(\mathbf{x})$ be the underlying data distribution, from which we have a finite number of iid samples but do not know its pdf. The **score matching** objective [Hyv05] minimizes a discrepancy between two distributions called the **Fisher divergence**:

$$D_F(p_{\mathcal{D}}(\mathbf{x}) \parallel p_{\boldsymbol{\theta}}(\mathbf{x})) = \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}\left[\frac{1}{2}\left\|\nabla_{\mathbf{x}} \log p_{\mathcal{D}}(\mathbf{x}) - \nabla_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\mathbf{x})\right\|^2\right]. \tag{24.28}$$

The expectation wrt $p_{\mathcal{D}}(\mathbf{x})$, in this objective and its variants below, admits a trivial unbiased Monte Carlo estimator using the empirical mean of samples $\mathbf{x} \sim p_{\mathcal{D}}(\mathbf{x})$. However, the second term of Equation (24.28), $\nabla_{\mathbf{x}} \log p_{\mathcal{D}}(\mathbf{x})$, is generally impractical to calculate since it requires knowing the pdf of $p_{\mathcal{D}}(\mathbf{x})$. We discuss a solution to this below.

24.3.1 Basic score matching

Hyvärinen [Hyv05] shows that, under certain regularity conditions, the Fisher divergence can be rewritten using integration by parts, with second derivatives of $\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})$ replacing the unknown first derivatives of $p_{\mathcal{D}}(\mathbf{x})$:

$$D_F(p_{\mathcal{D}}(\mathbf{x}) \parallel p_{\boldsymbol{\theta}}(\mathbf{x})) = \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}\left[\frac{1}{2}\sum_{i=1}^{d}\left(\frac{\partial \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}{\partial x_i}\right)^2 - \frac{\partial^2 \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}{\partial x_i^2}\right] + \text{constant} \tag{24.29}$$

$$= \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}\left[\frac{1}{2}||\boldsymbol{s}_{\boldsymbol{\theta}}(\mathbf{x})||^2 + \text{tr}(\mathbf{J}_{\mathbf{x}}\boldsymbol{s}_{\boldsymbol{\theta}}(\mathbf{x}))\right] + \text{constant} \tag{24.30}$$

where d is the dimensionality of $\mathbf{x}$, and $\mathbf{J}_{\mathbf{x}} s_{\boldsymbol{\theta}}(\mathbf{x})$ is the Jacobian of the score function. The constant does not affect optimization and thus can be dropped for training. It is shown by [Hyv05] that estimators based on score matching are consistent under some regularity conditions, meaning that the parameter estimator obtained by minimizing Equation (24.28) converges to the true parameters in the limit of infinite data. See Figure 25.5 for an example.

An important downside of the objective Equation (24.30) is that it takes $O(d^2)$ time to compute the trace of the Jacobian. For this reason, the implicit SM formulation of Equation (24.30) has only

been applied to relatively simple energy functions where computation of the second derivatives is tractable.

Score Matching assumes a continuous data distribution with positive density over the space, but it can be generalized to discrete or bounded data distributions [Hyv07b; Lyu12]. It is also possible to consider higher-order gradients of log pdf's beyond first derivatives [PDL+12].

24.3.2 Denoising score matching (DSM)

The Score Matching objective in Equation (24.30) requires several regularity conditions for $\log p_{\mathcal{D}}(\mathbf{x})$, e.g., it should be continuously differentiable and finite everywhere. However, these conditions may not always hold in practice. For example, a distribution of digital images is typically discrete and bounded, because the values of pixels are restricted to the range $\{0, 1, \cdots, 255\}$. Therefore, $\log p_{\mathcal{D}}(\mathbf{x})$ in this case is discontinuous and is negative infinity outside the range, and thus SM is not directly applicable.

To alleviate this, one can add a bit of noise to each datapoint: $\tilde{\mathbf{x}} = \mathbf{x} + \boldsymbol{\epsilon}$. As long as the noise distribution $p(\boldsymbol{\epsilon})$ is smooth, the resulting noisy data distribution $q(\tilde{\mathbf{x}}) = \int q(\tilde{\mathbf{x}} \mid \mathbf{x}) p_{\mathcal{D}}(\mathbf{x}) d\mathbf{x}$ is also smooth, and thus the Fisher divergence $D_F(q(\tilde{\mathbf{x}}) \parallel p_{\boldsymbol{\theta}}(\tilde{\mathbf{x}}))$ is a proper objective. [KL10] showed that the objective with noisy data can be approximated by the noiseless Score Matching objective of Equation (24.30) plus a regularization term; this regularization makes Score Matching applicable to a wider range of data distributions, but still requires expensive second-order derivatives.

[Vin11] proposed an elegant and scalable solution to the above difficulty, by showing that:

$$D_F(q(\tilde{\mathbf{x}}) \parallel p_{\boldsymbol{\theta}}(\tilde{\mathbf{x}})) = \mathbb{E}_{q(\tilde{\mathbf{x}})}\left[\frac{1}{2}\left\|\nabla_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\tilde{\mathbf{x}}) - \nabla_{\mathbf{x}} \log q(\tilde{\mathbf{x}})\right\|_2^2\right] \tag{24.31}$$

$$= \mathbb{E}_{q(\mathbf{x},\tilde{\mathbf{x}})}\left[\frac{1}{2}\left\|\nabla_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\tilde{\mathbf{x}}) - \nabla_{\mathbf{x}} \log q(\tilde{\mathbf{x}}|\mathbf{x})\right\|_2^2\right] + \text{constant} \tag{24.32}$$

$$= \frac{1}{2}\mathbb{E}_{q(\mathbf{x},\tilde{\mathbf{x}})}\left[\left\|\boldsymbol{s}_{\boldsymbol{\theta}}(\tilde{\mathbf{x}}) + \frac{(\tilde{\mathbf{x}} - \boldsymbol{x})}{\sigma^2}\right\|_2^2\right] \tag{24.33}$$

where $\boldsymbol{s}_{\boldsymbol{\theta}}(\tilde{\mathbf{x}}) = \nabla_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\tilde{\mathbf{x}})$ is the estimated score function, and

$$\nabla_{\mathbf{x}} \log q(\tilde{\mathbf{x}}|\mathbf{x}) = \nabla_{\mathbf{x}} \log \mathcal{N}(\tilde{\mathbf{x}}|\mathbf{x}, \sigma^2\mathbf{I}) = \frac{(\tilde{\mathbf{x}} - \boldsymbol{x})}{\sigma^2} \tag{24.34}$$

The expectation can be approximated by sampling from $p_{\mathcal{D}}(\mathbf{x})$ and then sampling the noise term $\tilde{\mathbf{x}}$. (The constant term does not affect optimization and can be ignored without changing the optimal solution.)

This estimation method is called **denoising score matching** (DSM) by [Vin11]. Similar formulations were also explored by Raphan and Simoncelli [RS07; RS11] and can be traced back to Tweedie's formula (Supplementary Section 3.3) and Stein's unbiased risk estimation [Ste81].

24.3.2.1 Difficulties

The major drawback of adding noise to data arises when $p_{\mathcal{D}}(\mathbf{x})$ is already a well-behaved distribution that satisfies the regularity conditions required by score matching. In this case, $D_F(q(\tilde{\mathbf{x}}) \parallel p_{\boldsymbol{\theta}}(\tilde{\mathbf{x}})) \neq$

$D_F(p_{\mathcal{D}}(\mathbf{x}) \parallel p_{\boldsymbol{\theta}}(\mathbf{x}))$, and DSM is not a consistent objective because the optimal EBM matches the noisy distribution $q(\tilde{\mathbf{x}})$, not $p_{\mathcal{D}}(\mathbf{x})$. This inconsistency becomes non-negligible when $q(\tilde{\mathbf{x}})$ significantly differs from $p_{\mathcal{D}}(\mathbf{x})$.

One way to attenuate the inconsistency of DSM is to choose $q \approx p_{\mathcal{D}}$, i.e., use a small noise perturbation. However, this often significantly increases the variance of objective values and hinders optimization. As an example, suppose $q(\tilde{\mathbf{x}} \mid \mathbf{x}) = \mathcal{N}(\tilde{\mathbf{x}} \mid \mathbf{x}, \sigma^2 I)$ and $\sigma \approx 0$. The corresponding DSM objective is

$$\begin{aligned} D_F(q(\tilde{\mathbf{x}}) \parallel p_{\boldsymbol{\theta}}(\tilde{\mathbf{x}})) &= \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} \mathbb{E}_{\mathbf{z} \sim \mathcal{N}(0,I)} \left[\frac{1}{2} \left\| \frac{\mathbf{z}}{\sigma} + \nabla_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\mathbf{x} + \sigma \mathbf{z}) \right\|_2^2 \right] \\ &\simeq \frac{1}{2N} \sum_{i=1}^{N} \left\| \frac{\mathbf{z}^{(i)}}{\sigma} + \nabla_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)} + \sigma \mathbf{z}^{(i)}) \right\|_2^2, \end{aligned} \tag{24.35}$$

where $\{\mathbf{x}^{(i)}\}_{i=1}^N \overset{\text{i.i.d.}}{\sim} p_{\mathcal{D}}(\mathbf{x})$, and $\{\mathbf{z}^{(i)}\}_{i=1}^N \overset{\text{i.i.d.}}{\sim} \mathcal{N}(\mathbf{0}, \boldsymbol{I})$. When $\sigma \to 0$, we can leverage Taylor series expansion to rewrite the Monte Carlo estimator in Equation (24.35) to

$$\frac{1}{2N} \sum_{i=1}^{N} \left[\frac{2}{\sigma} (\mathbf{z}^{(i)})^{\mathsf{T}} \nabla_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) + \frac{\left\| \mathbf{z}^{(i)} \right\|_2^2}{\sigma^2} \right] + \text{constant}. \tag{24.36}$$

When estimating the above expectation with samples, the variances of $(\mathbf{z}^{(i)})^{\mathsf{T}} \nabla_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})/\sigma$ and $\left\| \mathbf{z}^{(i)} \right\|_2^2 / \sigma^2$ will both grow unbounded as $\sigma \to 0$ due to division by σ and σ^2. This enlarges the variance of DSM and makes optimization challenging. Various methods have been proposed to reduce this variance (see e.g., [Wan+20d]).

24.3.3 Sliced score matching (SSM)

By adding noise to data, DSM avoids the expensive computation of second-order derivatives. However, as mentioned before, the optimal EBM that minimizes the DSM objective corresponds to the distribution of noise-perturbed data $q(\tilde{\mathbf{x}})$, not the original noise-free data distribution $p_{\mathcal{D}}(\mathbf{x})$. In other words, DSM does not give a consistent estimator of the data distribution, i.e., one cannot directly obtain an EBM that exactly matches the data distribution even with unlimited data.

Sliced score matching (SSM) [Son+19] is one alternative to Denoising Score Matching that is both consistent and computationally efficient. Instead of minimizing the Fisher divergence between two vector-valued scores, SSM randomly samples a projection vector $\mathbf{v}$, takes the inner product between $\mathbf{v}$ and the two scores, and then compares the resulting two scalars. More specifically, sliced ccore matching minimizes the following divergence called the **sliced Fisher divergence**:

$$D_{SF}(p_{\mathcal{D}}(\mathbf{x}) || p_{\boldsymbol{\theta}}(\mathbf{x})) = \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} \mathbb{E}_{p(\mathbf{v})} \left[\frac{1}{2} (\mathbf{v}^{\mathsf{T}} \nabla_{\mathbf{x}} \log p_{\mathcal{D}}(\mathbf{x}) - \mathbf{v}^{\mathsf{T}} \nabla_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\mathbf{x}))^2 \right], \tag{24.37}$$

where $p(\mathbf{v})$ denotes a projection distribution such that $\mathbb{E}_{p(\mathbf{v})}[\mathbf{v}\mathbf{v}^{\mathsf{T}}]$ is positive definite. Similar to Fisher divergence, sliced Fisher divergence has an implicit form that does not involve the unknown $\nabla_{\mathbf{x}} \log p_{\mathcal{D}}(\mathbf{x})$, which is given by

$$D_{SF}(p_{\mathcal{D}}(\mathbf{x}) \| p_{\boldsymbol{\theta}}(\mathbf{x})) = \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})} \mathbb{E}_{p(\mathbf{v})} \left[\frac{1}{2} \sum_{i=1}^{d} \left(\frac{\partial \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}{\partial x_i} v_i \right)^2 + \sum_{i=1}^{d} \sum_{j=1}^{d} \frac{\partial^2 \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}{\partial x_i \partial x_j} v_i v_j \right] + C. \tag{24.38}$$

All expectations in the above objective can be estimated with empirical means, and again the constant term C can be removed without affecting training. The second term involves second-order derivatives of $\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})$, but contrary to SM, it can be computed efficiently with a cost linear in the dimensionality d. This is because

$$\sum_{i=1}^{d}\sum_{j=1}^{d}\frac{\partial^2\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}{\partial x_i\partial x_j}v_iv_j=\sum_{i=1}^{d}\frac{\partial}{\partial x_i}\underbrace{\left(\sum_{j=1}^{d}\frac{\partial\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}{\partial x_j}v_j\right)}_{:=f(\mathbf{x})}v_i, \tag{24.39}$$

where $f(\mathbf{x})$ is the same for different values of i. Therefore, we only need to compute it once with $O(d)$ computation, *plus* another $O(d)$ computation for the outer sum to evaluate Equation (24.39), whereas the original SM objective requires $O(d^2)$ computation.

For many choices of $p(\mathbf{v})$, part of the SSM objective (Equation (24.38)) can be evaluated in closed form, potentially leading to lower variance. For example, when $p(\mathbf{v})=\mathcal{N}(\mathbf{0},\boldsymbol{I})$, we have

$$\mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}\mathbb{E}_{p(\mathbf{v})}\left[\frac{1}{2}\sum_{i=1}^{d}\left(\frac{\partial\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}{\partial x_i}v_i\right)^2\right]=\mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}\left[\frac{1}{2}\sum_{i=1}^{d}\left(\frac{\partial\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}{\partial x_i}\right)^2\right] \tag{24.40}$$

and as a result,

$$D_{SF}(p_{\mathcal{D}}(\mathbf{x})\|p_{\boldsymbol{\theta}}(\mathbf{x}))=\mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}\mathbb{E}_{\mathbf{v}\sim\mathcal{N}(\mathbf{0},\boldsymbol{I})}\left[\frac{1}{2}\sum_{i=1}^{d}\left(\frac{\partial\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}{\partial x_i}\right)^2+\sum_{i=1}^{d}\sum_{j=1}^{d}\frac{\partial^2\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}{\partial x_i\partial x_j}v_iv_j\right]+C \tag{24.41}$$

$$=\mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}\mathbb{E}_{\mathbf{v}\sim\mathcal{N}(\mathbf{0},\boldsymbol{I})}\left[\frac{1}{2}(\boldsymbol{v}^{\mathsf{T}}\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x}))^2+\boldsymbol{v}^{\mathsf{T}}[\mathbf{J}\boldsymbol{v}]\right] \tag{24.42}$$

where $\mathbf{J}=\mathbf{J}_x\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x})$. (Note that $\mathbf{J}\boldsymbol{v}$ can be computed using a Jacobian vector product operation.)

The above objective Equation (24.41) can also be obtained by approximating the sum of second-order gradients in the standard SM objective (Equation (24.30)) with the Hutchinson trace estimator [Ski89; Hut89; Mey+21]. It often (but not always) has lower variance than Equation (24.38), and can perform better in some applications [Son+19].

24.3.4 Connection to contrastive divergence

Though score matching and contrastive divergence (Section 24.2.2) are seemingly very different approaches, they are closely connected to each other. In fact, score matching can be viewed as a special instance of contrastive divergence in the limit of a particular MCMC sampler [Hyv07a]. Moreover, the Fisher divergence optimized by Score Matching is related to the derivative of KL divergence [Cov99], which is the underlying objective of Contrastive Divergence.

Contrastive divergence requires sampling from the EBM $\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})$, and one popular method for doing so is Langevin MCMC. Recall from Section 24.2.1 that given any initial datapoint $\mathbf{x}^0$, the Langevin MCMC method executes the following

$$\mathbf{x}^{k+1}\leftarrow\mathbf{x}^k-\frac{\epsilon}{2}\nabla_{\mathbf{x}}\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x}^k)+\sqrt{\epsilon}\,\mathbf{z}^k, \tag{24.43}$$

iteratively for $k = 0, 1, \cdots, K-1$, where $\mathbf{z}^k \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ and $\epsilon > 0$ is the step size.

Suppose we only run one-step Langevin MCMC for contrastive divergence. In this case, the gradient of the log-likelihood is given by

$$\begin{aligned}\mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}[\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}}(\mathbf{x})] &= -\mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}[\nabla_{\boldsymbol{\theta}} \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})] + \mathbb{E}_{\mathbf{x} \sim p_{\boldsymbol{\theta}}(\mathbf{x})}\left[\nabla_{\boldsymbol{\theta}} \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})\right] \\ &\simeq -\mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}[\nabla_{\boldsymbol{\theta}} \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})] + \mathbb{E}_{p_{\boldsymbol{\theta}}(\mathbf{x}), \mathbf{z} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})}\left[\nabla_{\boldsymbol{\theta}} \mathcal{E}_{\boldsymbol{\theta}}\left(\mathbf{x} - \frac{\epsilon^2}{2} \nabla_{\mathbf{x}} E_{\boldsymbol{\theta}'}(\mathbf{x}) + \epsilon\, \mathbf{z}\right)\Big|_{\boldsymbol{\theta}'=\boldsymbol{\theta}}\right]. \quad (24.44)\end{aligned}$$

After Taylor series expansion with respect to ϵ followed by some algebraic manipulations, the above equation can be transformed to the following [Hyv07a]:

$$\frac{\epsilon^2}{2} \nabla_{\boldsymbol{\theta}} D_F(p_{\mathcal{D}}(\mathbf{x}) \parallel p_{\boldsymbol{\theta}}(\mathbf{x})) + o(\epsilon^2). \quad (24.45)$$

When ϵ is sufficiently small, it corresponds to the re-scaled gradient of the score matching objective.

In general, score matching minimizes the Fisher divergence $D_F(p_{\mathcal{D}}(\mathbf{x}) \parallel p_{\boldsymbol{\theta}}(\mathbf{x}))$, whereas Contrastive Divergence minimizes an objective related to the KL divergence $D_{KL}(p_{\mathcal{D}}(\mathbf{x}) \parallel p_{\boldsymbol{\theta}}(\mathbf{x}))$, as shown in Equation (24.19). The above connection of score matching and Contrastive Divergence is a natural consequence of the connection between those two statistical divergences, as characterized by *de Bruijin's identity* [Cov99; Lyu12]:

$$\frac{d}{dt} D_{KL}(q_t(\tilde{\mathbf{x}}) \parallel p_{\boldsymbol{\theta},t}(\tilde{\mathbf{x}})) = -\frac{1}{2} D_F(q_t(\tilde{\mathbf{x}}) \parallel p_{\boldsymbol{\theta},t}(\tilde{\mathbf{x}})).$$

Here $q_t(\tilde{\mathbf{x}})$ and $p_{\boldsymbol{\theta},t}(\tilde{\mathbf{x}})$ denote smoothed versions of $p_{\mathcal{D}}(\mathbf{x})$ and $p_{\boldsymbol{\theta}}(\mathbf{x})$, resulting from adding Gaussian noise to $\mathbf{x}$ with variance t; i.e., $\tilde{\mathbf{x}} \sim \mathcal{N}(\mathbf{x}, t\boldsymbol{I})$.

24.3.5 Score-based generative models

We have seen how to use score matching to fit EBMs by learning the scalar energy function $\mathcal{E}_{\boldsymbol{\theta}}(\boldsymbol{x})$. We can alternatively directly learn the score function, $\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x}) = \nabla_{\boldsymbol{x}} \log p_{\boldsymbol{\theta}}(\boldsymbol{x})$; this is called a score-based generative model, and is discussed in Section 25.3. Such unconstrained score models are not guaranteed to output a conservative vector field, meaning they do not correspond to the gradient of any function. However, both methods seem to give comparable results [SH21].

24.4 Noise contrastive estimation

Another principle for learning the parameters of EBMs is **Noise contrastive estimation** (NCE), introduced by [GH10]. It is based on the idea that we can learn an EBM by contrasting it with another distribution with known density.

Let $p_{\mathcal{D}}(\mathbf{x})$ be our data distribution, and let $p_{\mathrm{n}}(\mathbf{x})$ be a chosen distribution with known density, called a noise distribution. This noise distribution is usually simple and has a tractable pdf, like $\mathcal{N}(\mathbf{0}, \boldsymbol{I})$, such that we can compute the pdf and generate samples from it efficiently. Strategies exist to learn the noise distribution, as referenced below. Furthermore, let y be a binary variable with Bernoulli distribution, which we use to define a mixture distribution of noise and data: $p_{\mathrm{n,data}}(\mathbf{x}) =$

$p(y=0)p_{\mathrm{n}}(\mathbf{x}) + p(y=1)p_{\mathcal{D}}(\mathbf{x})$. According to Bayes' rule, given a sample $\mathbf{x}$ from this mixture, the posterior probability of $y=0$ is

$$p_{\mathrm{n,data}}(y=0 \mid \mathbf{x}) = \frac{p_{\mathrm{n,data}}(\mathbf{x} \mid y=0)p(y=0)}{p_{\mathrm{n,data}}(\mathbf{x})} = \frac{p_{\mathrm{n}}(\mathbf{x})}{p_{\mathrm{n}}(\mathbf{x}) + \nu p_{\mathcal{D}}(\mathbf{x})} \tag{24.46}$$

where $\nu = p(y=1)/p(y=0)$.

Let our EBM $p_{\boldsymbol{\theta}}(\mathbf{x})$ be defined as:

$$p_{\boldsymbol{\theta}}(\mathbf{x}) = \exp(-\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x}))/Z_{\boldsymbol{\theta}} \tag{24.47}$$

Contrary to most other EBMs, $Z_{\boldsymbol{\theta}}$ is treated as a learnable (scalar) parameter in NCE. Given this model, similar to the mixture of noise and data above, we can define a mixture of noise and the model distribution: $p_{\mathrm{n},\boldsymbol{\theta}}(\mathbf{x}) = p(y=0)p_{\mathrm{n}}(\mathbf{x}) + p(y=1)p_{\boldsymbol{\theta}}(\mathbf{x})$. The posterior probability of $y=0$ given this noise/model mixture is:

$$p_{\mathrm{n},\boldsymbol{\theta}}(y=0 \mid \mathbf{x}) = \frac{p_{\mathrm{n}}(\mathbf{x})}{p_{\mathrm{n}}(\mathbf{x}) + \nu p_{\boldsymbol{\theta}}(\mathbf{x})} \tag{24.48}$$

In NCE, we indirectly fit $p_{\boldsymbol{\theta}}(\mathbf{x})$ to $p_{\mathcal{D}}(\mathbf{x})$ by fitting $p_{\mathrm{n},\boldsymbol{\theta}}(y \mid \mathbf{x})$ to $p_{\mathrm{n,data}}(y \mid \mathbf{x})$ through a standard conditional maximum likelihood objective:

$$\boldsymbol{\theta}^* = \operatorname*{argmin}_{\boldsymbol{\theta}} \mathbb{E}_{p_{\mathrm{n,data}}(\mathbf{x})}[D_{KL}(p_{\mathrm{n,data}}(y \mid \mathbf{x}) \parallel p_{\mathrm{n},\boldsymbol{\theta}}(y \mid \mathbf{x}))] \tag{24.49}$$

$$= \operatorname*{argmax}_{\boldsymbol{\theta}} \mathbb{E}_{p_{\mathrm{n,data}}(\mathbf{x},y)}[\log p_{\mathrm{n},\boldsymbol{\theta}}(y \mid \mathbf{x})], \tag{24.50}$$

which can be solved using stochastic gradient ascent. Just like any other deep classifier, when the model is sufficiently powerful, $p_{\mathrm{n},\boldsymbol{\theta}^*}(y \mid \mathbf{x})$ will match $p_{\mathrm{n,data}}(y \mid \mathbf{x})$ at the optimum. In that case:

$$p_{\mathrm{n},\boldsymbol{\theta}^*}(y=0 \mid \mathbf{x}) \equiv p_{\mathrm{n,data}}(y=0 \mid \mathbf{x}) \tag{24.51}$$

$$\iff \frac{p_{\mathrm{n}}(\mathbf{x})}{p_{\mathrm{n}}(\mathbf{x}) + \nu p_{\boldsymbol{\theta}^*}(\mathbf{x})} \equiv \frac{p_{\mathrm{n}}(\mathbf{x})}{p_{\mathrm{n}}(\mathbf{x}) + \nu p_{\mathcal{D}}(\mathbf{x})} \tag{24.52}$$

$$\iff p_{\boldsymbol{\theta}^*}(\mathbf{x}) \equiv p_{\mathcal{D}}(\mathbf{x}) \tag{24.53}$$

Consequently, $E_{\boldsymbol{\theta}^*}(\mathbf{x})$ is an unnormalized energy function that matches the data distribution $p_{\mathcal{D}}(\mathbf{x})$, and $Z_{\boldsymbol{\theta}^*}$ is the corresponding normalizing constant.

As one unique feature that contrastive divergence and score matching do not have, NCE provides the normalizing constant of an Energy-Based Model as a by-product of its training procedure. When the EBM is very expressive, e.g., a deep neural network with many parameters, we can assume it is able to approximate a normalized probability density and absorb $Z_{\boldsymbol{\theta}}$ into the parameters of $\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})$ [MT12], or equivalently, fixing $Z_{\boldsymbol{\theta}} = 1$. The resulting EBM trained with NCE will be self-normalized, i.e., having a normalizing constant close to 1.

In practice, choosing the right noise distribution $p_{\mathrm{n}}(\mathbf{x})$ is critical to the success of NCE, especially for structured and high-dimensional data. As argued in Gutmann and Hirayama [GH12], NCE works the best when the noise distribution is close to the data distribution (but not exactly the same). Many methods have been proposed to automatically tune the noise distribution, such as Adversarial Contrastive Estimation [BLC18], Conditional NCE [CG18] and Flow Contrastive Estimation [Gao+20]. NCE can be further generalized using Bregman divergences (Section 5.1.10), where the formulation introduced here reduces to a special case.

24.4.1 Connection to score matching

Noise contrastive estimation provides a family of objectives that vary for different $p_{\mathrm{n}}(\mathbf{x})$ and ν. This flexibility may allow adaptation to special properties of a task with hand-tuned $p_{\mathrm{n}}(\mathbf{x})$ and ν, and may also give a unified perspective for different approaches. In particular, when using an appropriate $p_{\mathrm{n}}(\mathbf{x})$ and a slightly different parameterization of $p_{\mathrm{n},\boldsymbol{\theta}}(y \mid \mathbf{x})$, we can recover score matching from NCE [GH12].

Specifically, we choose the noise distribution $p_{\mathrm{n}}(\mathbf{x})$ to be a perturbed data distribution: given a small (deterministic) vector $\mathbf{v}$, let $p_{\mathrm{n}}(\mathbf{x}) = p_{\mathcal{D}}(\mathbf{x} - \mathbf{v})$. It is efficient to sample from this $p_{\mathrm{n}}(\mathbf{x})$, since we can first draw any datapoint $\mathbf{x}' \sim p_{\mathcal{D}}(\mathbf{x}')$ and then compute $\mathbf{x} = \mathbf{x}' + \mathbf{v}$. It is, however, difficult to evaluate the density of $p_{\mathrm{n}}(\mathbf{x})$ because $p_{\mathcal{D}}(\mathbf{x})$ is unknown. Since the original parameterization of $p_{\mathrm{n},\boldsymbol{\theta}}(y \mid \mathbf{x})$ in NCE (Equation (24.48)) depends on the pdf of $p_{\mathrm{n}}(\mathbf{x})$, we cannot directly apply the standard NCE objective. Instead, we replace $p_{\mathrm{n}}(\mathbf{x})$ with $p_{\boldsymbol{\theta}}(\mathbf{x} - \mathbf{v})$ and parameterize $p_{\mathrm{n},\boldsymbol{\theta}}(y = 0 \mid \mathbf{x})$ with the following form

$$p_{\mathrm{n},\boldsymbol{\theta}}(y = 0 \mid \mathbf{x}) := \frac{p_{\boldsymbol{\theta}}(\mathbf{x} - \mathbf{v})}{p_{\boldsymbol{\theta}}(\mathbf{x}) + p_{\boldsymbol{\theta}}(\mathbf{x} - \mathbf{v})} \tag{24.54}$$

In this case, the NCE objective (Equation (24.50)) reduces to:

$$\boldsymbol{\theta}^* = \operatorname*{argmin}_{\boldsymbol{\theta}} \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}[\log(1 + \exp(\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x}) - \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x} - \mathbf{v})) + \log(1 + \exp(\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x}) - \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x} + \mathbf{v}))] \tag{24.55}$$

At $\boldsymbol{\theta}^*$, we have a solution where:

$$p_{\mathrm{n},\boldsymbol{\theta}^*}(y = 0 \mid \mathbf{x}) \equiv p_{\mathrm{n},\mathrm{data}}(y = 0 \mid \mathbf{x}) \tag{24.56}$$

$$\implies \frac{p_{\boldsymbol{\theta}^*}(\mathbf{x} - \mathbf{v})}{p_{\boldsymbol{\theta}^*}(\mathbf{x}) + p_{\boldsymbol{\theta}^*}(\mathbf{x} - \mathbf{v})} \equiv \frac{p_{\mathcal{D}}(\mathbf{x} - \mathbf{v})}{p_{\mathcal{D}}(\mathbf{x}) + p_{\mathcal{D}}(\mathbf{x} - \mathbf{v})} \tag{24.57}$$

which implies that $p_{\boldsymbol{\theta}^*}(\mathbf{x}) \equiv p_{\mathcal{D}}(\mathbf{x})$, i.e., our model matches the data distribution.

As noted in Gutmann and Hirayama [GH12] and Song et al. [Son+19], when $\|\mathbf{v}\|_2 \approx 0$, the NCE objective Equation (24.50) has the following equivalent form by Taylor expansion

$$\operatorname*{argmin}_{\boldsymbol{\theta}} \frac{1}{4}\mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}\left[\frac{1}{2}\sum_{i=1}^{d}\left(\frac{\partial \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}{\partial x_i}v_i\right)^2 + \sum_{i=1}^{d}\sum_{j=1}^{d}\frac{\partial^2 \mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}{\partial x_i \partial x_j}v_i v_j\right] + 2\log 2 + o(\|\mathbf{v}\|_2^2). \tag{24.58}$$

Comparing against Equation (24.38), we immediately see that the above objective equals that of SSM, if we ignore small additional terms hidden in $o(\|\mathbf{v}\|_2^2)$ and take the expectation with respect to $\mathbf{v}$ over a user-specified distribution $p(\mathbf{v})$.

24.5 Other methods

Aside from MCMC-based training, score matching and noise contrastive estimation, there are also other methods for learning EBMs. Below we briefly survey some examples of them. Interested readers can learn more details from references therein.

24.5.1 Minimizing Differences/Derivatives of KL Divergences

The overarching strategy for learning probabilistic models from data is to minimize the KL divergence between data and model distributions. However, because the normalizing constants of EBMs are typically intractable, it is hard to directly evaluate the KL divergence when the model is an EBM (see the discussion in Section 24.2.1). One generic idea that has frequently circumvented this difficulty is to consider differences/derivatives of KL divergences. It turns out that the unknown partition functions of EBMs are often cancelled out after taking the difference of two closely related KL divergences, or computing the derivatives.

Typical examples of this strategy include minimum velocity learning [Mov08; Wan+20d], minimum probability flow [SDBD11], and minimum KL contraction [Lyu11], to name a few. In minimum velocity learning and minimum probability flow, a Markov chain is designed such that it starts from the data distribution $p_{\mathcal{D}}(\mathbf{x})$ and converges to the EBM distribution $p_{\boldsymbol{\theta}}(\mathbf{x}) = e^{-\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}/Z_{\boldsymbol{\theta}}$. Specifically, the Markov chain satisfies $p_0(\mathbf{x}) \equiv p_{\mathcal{D}}(\mathbf{x})$ and $p_\infty(\mathbf{x}) \equiv p_{\boldsymbol{\theta}}(\mathbf{x})$, where we denote by $p_t(\mathbf{x})$ the state distribution at time $t \geq 0$.

This Markov chain will evolve towards $p_{\boldsymbol{\theta}}(\mathbf{x})$ unless $p_{\mathcal{D}}(\mathbf{x}) \equiv p_{\boldsymbol{\theta}}(\mathbf{x})$. Therefore, we can fit the EBM distribution $p_{\boldsymbol{\theta}}(\mathbf{x})$ to $p_{\mathcal{D}}(\mathbf{x})$ by minimizing the modulus of the "velocity" of this evolution, defined by

$$\frac{\mathrm{d}}{\mathrm{d}t} \mathrm{D}_{\mathbb{KL}}(p_t(\mathbf{x}) \parallel p_{\boldsymbol{\theta}}(\mathbf{x}))\Big|_{t=0} \quad \text{or} \quad \frac{\mathrm{d}}{\mathrm{d}t} \mathrm{D}_{\mathbb{KL}}(p_{\mathcal{D}}(\mathbf{x}) \parallel p_t(\mathbf{x}))\Big|_{t=0} \tag{24.59}$$

in minimum velocity learning and minimum probability flow respectively. These objectives typically do not require computing the normalizing constant $Z_{\boldsymbol{\theta}}$.

In minimum KL contraction [Lyu11], a distribution transformation Φ is chosen such that

$$\mathrm{D}_{\mathbb{KL}}(p(\mathbf{x}) \parallel q(\mathbf{x})) \geq \mathrm{D}_{\mathbb{KL}}(\Phi\{p(\mathbf{x})\} \parallel \Phi\{q(\mathbf{x})\}) \tag{24.60}$$

with equality if and only if $p(\mathbf{x}) = q(\mathbf{x})$. We can leverage this Φ to train an EBM, by minimizing

$$\mathrm{D}_{\mathbb{KL}}(p_{\mathcal{D}}(\mathbf{x}) \parallel p_{\boldsymbol{\theta}}(\mathbf{x})) - \mathrm{D}_{\mathbb{KL}}(\Phi\{p_{\mathcal{D}}(\mathbf{x})\} \parallel \Phi\{p_{\boldsymbol{\theta}}(\mathbf{x})\}). \tag{24.61}$$

This objective does not require computing the partition function $Z_{\boldsymbol{\theta}}$ whenever Φ is linear.

Minimum velocity learning, minimum probability flow, and minimum KL contraction can all be viewed as generalizations to score matching and noise contrastive estimation [Mov08; SDBD11; Lyu11].

24.5.2 Minimizing the Stein discrepancy

We can train EBMs by minimizing the Stein discrepancy, defined by

$$D_{\text{Stein}}(p_{\mathcal{D}}(\mathbf{x}) \parallel p_{\boldsymbol{\theta}}(\mathbf{x})) := \sup_{\mathbf{f} \in \mathcal{F}} \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}[\nabla_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\mathbf{x})^{\mathsf{T}} \mathbf{f}(\mathbf{x}) + \operatorname{trace}(\nabla_{\mathbf{x}} \mathbf{f}(\mathbf{x}))], \tag{24.62}$$

where $\mathcal{F}$ is a family of vector-valued functions, and $\nabla_{\mathbf{x}} \mathbf{f}(\mathbf{x})$ denotes the Jacobian of $\mathbf{f}(\mathbf{x})$. With some regularity conditions [GM15; LLJ16], we have $D_S(p_{\mathcal{D}}(\mathbf{x}) \parallel p_{\boldsymbol{\theta}}(\mathbf{x})) \geq 0$, where the equality holds if and only if $p_{\mathcal{D}}(\mathbf{x}) \equiv p_{\boldsymbol{\theta}}(\mathbf{x})$. Similar to score matching (Equation (24.30)), the objective Equation (24.62) only involves the score function of $p_{\boldsymbol{\theta}}(\mathbf{x})$, and does not require computing the EBM's

partition function. Still, the trace term in Equation (24.62) may demand expensive computation, and does not scale well to high dimensional data.

There are two common methods that sidestep this difficulty. Gorham and Mackey [GM15] and Liu, Lee, and Jordan [LLJ16] discovered that when $\mathcal{F}$ is a unit ball in a reproducing kernel Hilbert space (RKHS) with a fixed kernel, the Stein discrepancy becomes kernelized Stein discrepancy, where the trace term is a constant and does not affect optimization. Otherwise, $\text{trace}(\nabla_{\mathbf{x}}\mathbf{f}(\mathbf{x}))$ can be approximated with the Skilling-Hutchinson trace estimator [Ski89; Hut89; Gra+20c].

24.5.3 Adversarial training

Recall from Section 24.2.1 that when training EBMs with maximum likelihood estimation (MLE), we need to sample from the EBM per training iteration. However, sampling using multiple MCMC steps is expensive and requires careful tuning of the Markov chain. One way to avoid this difficulty is to use non-MLE methods that do not need sampling, such as score matching and noise contrastive estimation. Here we introduce another family of methods that sidestep costly MCMC sampling by learning an auxiliary model through adversarial training, which allows fast sampling.

Using the definition of EBMs, we can rewrite the maximum likelihood objective by introducing a variational distribution $q_{\boldsymbol{\phi}}(\mathbf{x})$ parameterized by $\boldsymbol{\phi}$:

$$\begin{aligned}
\mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}[\log p_{\boldsymbol{\theta}}(\mathbf{x})] &= \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}[-\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})] - \log Z_{\boldsymbol{\theta}} \\
&= \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}[-\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})] - \log \int e^{-\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})} \mathrm{d}\mathbf{x} \\
&= \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}[-\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})] - \log \int q_{\boldsymbol{\phi}}(\mathbf{x}) \frac{e^{-\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}}{q_{\boldsymbol{\phi}}(\mathbf{x})} \mathrm{d}\mathbf{x} \\
&\overset{(i)}{\leq} \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}[-\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})] - \int q_{\boldsymbol{\phi}}(\mathbf{x}) \log \frac{e^{-\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})}}{q_{\boldsymbol{\phi}}(\mathbf{x})} \mathrm{d}\mathbf{x} \\
&= \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}[-\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})] - \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{x})}[-\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})] - H(q_{\boldsymbol{\phi}}(\mathbf{x})),
\end{aligned} \tag{24.63}$$

where $H(q_{\boldsymbol{\phi}}(\mathbf{x}))$ denotes the entropy of $q_{\boldsymbol{\phi}}(\mathbf{x})$. Step (i) is due to Jensen's inequality. Equation (24.63) provides an upper bound to the expected log-likelihood. For EBM training, we can first minimize the upper bound Equation (24.63) with respect to $q_{\boldsymbol{\phi}}(\mathbf{x})$ so that it is closer to the likelihood objective, and then maximize Equation (24.63) with respect to $\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})$ as a surrogate for maximizing likelihood. This amounts to using the following maximin objective

$$\max_{\boldsymbol{\theta}} \min_{\boldsymbol{\phi}} \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{x})}[\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})] - \mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}[\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})] - H(q_{\boldsymbol{\phi}}(\mathbf{x})). \tag{24.64}$$

Optimizing the above objective is similar to training GANs (Chapter 26), and can be achieved by adversarial training. The variational distribution $q_\phi(\mathbf{x})$ should allow both fast sampling and efficient entropy evaluation to make Equation (24.64) tractable. This limits the model family of $q_\phi(\mathbf{x})$, and usually restricts our choice to invertible probabilistic models, such as inverse autoregressive flow (Section 23.2.4.3). See Dai et al. [Dai+19b] for an example on designing $q_{\boldsymbol{\phi}}(\mathbf{x})$ and training EBMs with Equation (24.64).

Kim and Bengio [KB16] and Zhai et al. [Zha+16] propose to represent $q_{\boldsymbol{\phi}}(\mathbf{x})$ with neural samplers, like the generator of GANs. A neural sampler is a deterministic mapping $g_{\boldsymbol{\phi}}$ that maps a random

Gaussian noise $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ directly to a sample $\mathbf{x} = g_{\boldsymbol{\phi}}(\mathbf{z})$. When using a neural sampler as $q_{\boldsymbol{\phi}}(\mathbf{x})$, it is efficient to draw samples through the deterministic mapping, but $H(q_{\boldsymbol{\phi}}(\mathbf{x}))$ is intractable since the density of $q_{\boldsymbol{\phi}}(\mathbf{x})$ is unknown. Kim and Bengio [KB16] and Zhai et al. [Zha+16] propose several heuristics to approximate this entropy function. Kumar et al. [Kum+19c] propose to estimate the entropy through its connection to mutual information: $H(q_{\boldsymbol{\phi}}(\mathbf{z})) = I(g_{\boldsymbol{\phi}}(\mathbf{z}), \mathbf{z})$, which can be estimated from samples with variational lower bounds [NWJ10b; NCT16b]. Dai et al. [Dai+19a] noticed that when defining $p_{\boldsymbol{\theta}}(\mathbf{x}) = p_0(\mathbf{x}) e^{-\mathcal{E}_{\boldsymbol{\theta}}(\mathbf{x})} / Z_{\boldsymbol{\theta}}$, with $p_0(\mathbf{x})$ being a fixed base distribution, the entropy term $-H(q_{\boldsymbol{\phi}}(\mathbf{x}))$ in Equation (24.64) can be replaced by $\mathrm{D}_{\mathbb{KL}}(q_{\boldsymbol{\phi}}(\mathbf{x}) \parallel p_0(\mathbf{x}))$, which can also be approximated with variational lower bounds using samples from $q_{\boldsymbol{\phi}}(\mathbf{x})$ and $p_0(\mathbf{x})$, without requiring the density of $q_{\boldsymbol{\phi}}(\mathbf{x})$.

Grathwohl et al. [Gra+20a] represent $q_{\boldsymbol{\phi}}(\mathbf{x})$ as a noisy neural sampler, where samples are obtained via $g_{\boldsymbol{\phi}}(\mathbf{z}) + \sigma \boldsymbol{\epsilon}$, assuming $\mathbf{z}, \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$. With a noisy neural sampler, $\nabla_{\boldsymbol{\phi}} H(q_{\boldsymbol{\phi}}(\mathbf{x}))$ becomes particularly easy to estimate, which allows gradient-based optimization for the minimax objective in Equation (24.63). A related approach is proposed in Xie et al. [Xie+18], where authors train a noisy neural sampler with samples obtained from MCMC, and initialize new MCMC chains with samples generated from the neural sampler. This cooperative sampling scheme improves the convergence of MCMC, but may still require multiple MCMC steps for sample generation. It does not optimize the objective in Equation (24.63).

When using both adversarial training and MCMC sampling, Yu et al. [Yu+20] noticed that EBMs can be trained with an arbitrary f-divergence, including KL, reverse KL, total variation, Hellinger, etc. The method proposed by Yu et al. [Yu+20] allows us to explore the trade-offs and inductive bias of different statistical divergences for more flexible EBM training.

25 Diffusion models

25.1 Introduction

In this chapter, we consider a class of models called **diffusion models**. This class of models has recently generated a lot of interest, due to its ability to generate diverse, high quality, samples, and the relative simplicity of the training scheme, which allows very large models to be trained at scale. Diffusion models are closely related to VAEs (Chapter 21), normalizing flows (Chapter 23), and EBMs (Chapter 24), as we will see.

The basic idea behind these models is based on the observation that it is hard to convert noise into structured data, but it is easy to convert structured data into noise. In particular, we can use a **forwards process** or **diffusion process** to gradually convert the observed data $\boldsymbol{x}_0$ into a noisy version $\boldsymbol{x}_T$ by passing the data through T steps of a stochastic encoder $q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})$. After enough steps, we have $\boldsymbol{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, or some other convenient reference distribution. We then learn a **reverse process** to undo this, by passing the noise through T steps of a decoder $p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ until we generate $\boldsymbol{x}_0$. See Figure 25.1 for an overall sketch of the approach. In the following sections, we discuss this class of models in more detail. Our presentation is based in part on the excellent tutorial [KGV22]. More details can be found in the recent review papers [Yan+22; Cao+22], as well as specialized papers, such as [Kar+22]. There are also many other excellent resources online, such as https://github.com/heejkoo/Awesome-Diffusion-Models and https://scorebasedgenerativemodeling.github.io/. For a detailed tutorial on the underlying math, see [McA23].

25.2 Denoising diffusion probabilistic models (DDPMs)

In this section, we discuss **denoising diffusion probabilistic models** or **DDPMs**, introduced in [SD+15b], and then extended in [HJA20; Kin+21] and many other works. We can think of the DDPM as similar to a hierarchical variational autoencoder (Section 21.5), except that all the latent states (denoted $\boldsymbol{x}_t$ for $t = 1 : T$) have the same dimensionality as the input $\boldsymbol{x}_0$. (In this respect, a DDPM is similar to a normalizing flow (Chapter 23); however, in a diffusion model, the hidden layers are stochastic, and do not need to use invertible transformations.) In addition, the encoder network q is a simple linear Gaussian model, rather than being learned[1], and the decoder network p is shared across all time steps. These restrictions result in a very simple training objective, which

1. Later we will discuss some extensions in which the noise level of the encoder can also be learned. Nevertheless, the encoder remains simple, by design.

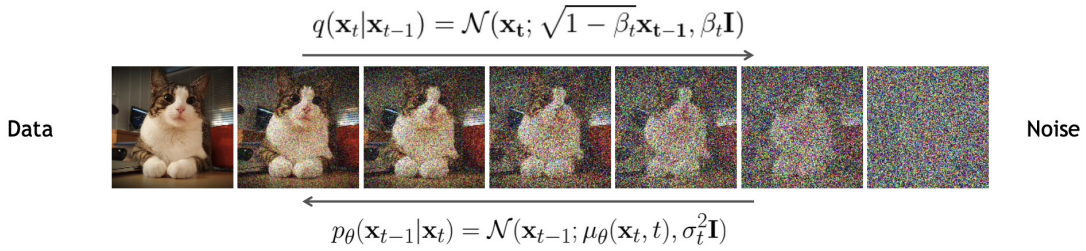


Figure 25.1: Denoising diffusion probabilistic model. The forwards diffusion process, $q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})$, implements the (non-learned) inference network; this just adds Gaussian noise at each step. The reverse diffusion process, $p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$, implements the decoder; this is a learned Gaussian model. From Slide 16 of [KGV22]. Used with kind permission of Arash Vahdat.

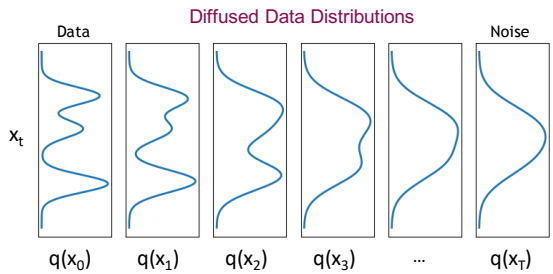


Figure 25.2: Illustration of a diffusion model on 1d data. The forwards diffusion process gradually transforms the empirical data distribution $q(\boldsymbol{x}_0)$ into a simple target distribution, here $q(\boldsymbol{x}_T) = \mathcal{N}(\mathbf{0}, \mathbf{I})$. To generate from the model, we sample a point $\boldsymbol{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, and then run the Markov chain backwards, by sampling $\boldsymbol{x}_t \sim p_{\boldsymbol{\theta}}(\boldsymbol{x}_t|\boldsymbol{x}_{t+1})$ until we get a sample in the original data space, $\boldsymbol{x}_0$. From Slide 19 of [KGV22]. Used with kind permission of Arash Vahdat.

allows deep models to be easily trained without any risk of posterior collapse (Section 21.4). In particular, in Section 25.2.3, we will see that training reduces to a series of weighted nonlinear least squares problems.

25.2.1 Encoder (forwards diffusion)

The forwards encoder process is defined to be a simple linear Gaussian model:

$$q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}) = \mathcal{N}(\boldsymbol{x}_t|\sqrt{1-\beta_t}\boldsymbol{x}_{t-1}, \beta_t\mathbf{I}) \tag{25.1}$$

where the values of $\beta_t \in (0, 1)$ are chosen according to a noise schedule (see Section 25.2.4). The joint distribution over all the latent states, conditioned on the input, is given by

$$q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0) = \prod_{t=1}^{T} q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}) \tag{25.2}$$

Since this defines a linear Gaussian Markov chain, we can compute marginals of it in closed form. In particular, we have

$$q(\boldsymbol{x}_t|\boldsymbol{x}_0) = \mathcal{N}(\boldsymbol{x}_t|\sqrt{\overline{\alpha}_t}\boldsymbol{x}_0, (1-\overline{\alpha}_t)\mathbf{I}) \tag{25.3}$$

where we define

$$\alpha_t \triangleq 1-\beta_t, \overline{\alpha}_t = \prod_{s=1}^{t} \alpha_s \tag{25.4}$$

We choose the noise schedule such that $\overline{\alpha}_T \approx 0$, so that $q(\boldsymbol{x}_T|\boldsymbol{x}_0) \approx \mathcal{N}(\mathbf{0}, \mathbf{I})$.

The distribution $q(\boldsymbol{x}_t|\boldsymbol{x}_0)$ is known as the **diffusion kernel**. Applying this to the input data distribution and then computing the result unconditional marginals is equivalent to Gaussian convolution:

$$q(\boldsymbol{x}_t) = \int q_0(\boldsymbol{x}_0) q(\boldsymbol{x}_t|\boldsymbol{x}_0) d\boldsymbol{x}_0 \tag{25.5}$$

As t increases, the marginals become simpler, as shown in Figure 25.2. In the image domain, this process will first remove high-frequency content (i.e., low-level details, such as texture), and later will remove low-frequency content (i.e., high-level or "semantic" information, such as shape), as shown in Figure 25.1.

25.2.2 Decoder (reverse diffusion)

In the reverse diffusion process, we would like to invert the forwards diffusion process. If we know the input $\boldsymbol{x}_0$, we can derive the reverse of one forwards step as follows:[2]

$$q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0) = \mathcal{N}(\boldsymbol{x}_{t-1}|\tilde{\mu}_t(\boldsymbol{x}_t, \boldsymbol{x}_0), \tilde{\beta}_t \mathbf{I}) \tag{25.6}$$

$$\tilde{\mu}_t(\boldsymbol{x}_t, \boldsymbol{x}_0) = \frac{\sqrt{\overline{\alpha}_{t-1}}\beta_t}{1-\overline{\alpha}_t}\boldsymbol{x}_0 + \frac{\sqrt{\alpha_t}(1-\overline{\alpha}_{t-1})}{1-\overline{\alpha}_t}\boldsymbol{x}_t \tag{25.7}$$

$$\tilde{\beta}_t = \frac{1-\overline{\alpha}_{t-1}}{1-\overline{\alpha}_t}\beta_t \tag{25.8}$$

Of course, when generating a new datapoint, we do not know $\boldsymbol{x}_0$, but we will train the generator to approximate the above distribution averaged over $\boldsymbol{x}_0$. Thus we choose the generator to have the form

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) = \mathcal{N}(\boldsymbol{x}_{t-1}|\boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t), \boldsymbol{\Sigma}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)) \tag{25.9}$$

We often set $\boldsymbol{\Sigma}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t) = \sigma_t^2 \mathbf{I}$. We discuss how to learn σ_t^2 in Section 25.2.4, but two natural choices are $\sigma_t^2 = \beta_t$ and $\sigma_t^2 = \tilde{\beta}_t$; these correspond to upper and lower bounds on the reverse process entropy, as shown in [HJA20].

The corresponding joint distribution over all the generated variables is given by $p_{\boldsymbol{\theta}}(\boldsymbol{x}_{0:T}) = p(\boldsymbol{x}_T)\prod_{t=1}^{T} p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$, where we set $p(\boldsymbol{x}_T) = \mathcal{N}(\mathbf{0}, \mathbf{I})$. We can sample from this distribution using the pseudocode in Algorithm 25.2.

2. We just need to use Bayes' rule for Gaussians. See e.g., https://lilianweng.github.io/posts/2021-07-11-diffusion-models/ for a detailed derivation.

25.2.3 Model fitting

We will fit the model by maximizing the evidence lower bound (ELBO), similar to how we train VAEs (see Section 21.2). In particular, for each data example $\boldsymbol{x}_0$ we have

$$\log p_{\boldsymbol{\theta}}(\boldsymbol{x}_0) = \log \left[\int d\boldsymbol{x}_{1:T} q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0) \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x}_{0:T})}{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)} \right] \tag{25.10}$$

$$\geq \int d\boldsymbol{x}_{1:T} q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0) \log \left(p(\boldsymbol{x}_T) \prod_{t=1}^{T} \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)}{q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})} \right) \tag{25.11}$$

$$= \mathbb{E}_q \left[\log p(\boldsymbol{x}_T) + \sum_{t=1}^{T} \log \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)}{q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})} \right] \triangleq Ł(\boldsymbol{x}_0) \tag{25.12}$$

We now discuss how to compute the terms in the ELBO. By the Markov property we have $q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}) = q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}, \boldsymbol{x}_0)$, and by Bayes' rule, we have

$$q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}, \boldsymbol{x}_0) = \frac{q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0) q(\boldsymbol{x}_t|\boldsymbol{x}_0)}{q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_0)} \tag{25.13}$$

Plugging Equation (25.13) into the ELBO we get

$$Ł(\boldsymbol{x}_0) = \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)} \left[\log p(\boldsymbol{x}_T) + \sum_{t=2}^{T} \log \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)}{q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0)} + \underbrace{\sum_{t=2}^{T} \log \frac{q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_0)}{q(\boldsymbol{x}_t|\boldsymbol{x}_0)}}_{*} + \log \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x}_0|\boldsymbol{x}_1)}{q(\boldsymbol{x}_1|\boldsymbol{x}_0)} \right] \tag{25.14}$$

The term marked * is a telescoping sum, and can be simplified as follows:

$$* = \log q(\boldsymbol{x}_{T-1}|\boldsymbol{x}_0) + \cdots + \log q(\boldsymbol{x}_2|\boldsymbol{x}_0) + \log q(\boldsymbol{x}_1|\boldsymbol{x}_0) \tag{25.15}$$

$$- \log q(\boldsymbol{x}_T|\boldsymbol{x}_0) - \log q(\boldsymbol{x}_{T-1}|\boldsymbol{x}_0) - \cdots - \log q(\boldsymbol{x}_2|\boldsymbol{x}_0) \tag{25.16}$$

$$= - \log q(\boldsymbol{x}_T|\boldsymbol{x}_0) + \log q(\boldsymbol{x}_1|\boldsymbol{x}_0) \tag{25.17}$$

Hence the negative ELBO (variational upper bound) becomes

$$\mathcal{L}(\boldsymbol{x}_0) = -\mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)} \left[\log \frac{p(\boldsymbol{x}_T)}{q(\boldsymbol{x}_T|\boldsymbol{x}_0)} + \sum_{t=2}^{T} \log \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)}{q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0)} + \log p_{\boldsymbol{\theta}}(\boldsymbol{x}_0|\boldsymbol{x}_1) \right] \tag{25.18}$$

$$= \underbrace{D_{\mathbb{KL}}\left(q(\boldsymbol{x}_T|\boldsymbol{x}_0) \parallel p(\boldsymbol{x}_T)\right)}_{L_T(\boldsymbol{x}_0)} \tag{25.19}$$

$$+ \sum_{t=2}^{T} \mathbb{E}_{q(\boldsymbol{x}_t|\boldsymbol{x}_0)} \underbrace{D_{\mathbb{KL}}\left(q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)\right)}_{L_{t-1}(\boldsymbol{x}_0)} - \underbrace{\mathbb{E}_{q(\boldsymbol{x}_1|\boldsymbol{x}_0)} \log p_{\boldsymbol{\theta}}(\boldsymbol{x}_0|\boldsymbol{x}_1)}_{L_0(\boldsymbol{x}_0)} \tag{25.20}$$

Each of these KL terms can be computed analytically, since all the distributions are Gaussian. Below we focus on the L_{t-1} term. Since $\boldsymbol{x}_t = \sqrt{\overline{\alpha}_t}\boldsymbol{x}_0 + \sqrt{(1-\overline{\alpha}_t)}\boldsymbol{\epsilon}$, we can write

$$\tilde{\boldsymbol{\mu}}_t(\boldsymbol{x}_t, \boldsymbol{x}_0) = \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t - \frac{\beta_t}{\sqrt{1-\overline{\alpha}_t}}\boldsymbol{\epsilon}\right) \tag{25.21}$$

Thus instead of training the model to predict the mean of the denoised version of $\boldsymbol{x}_{t-1}$ given its noisy input $\boldsymbol{x}_t$, we can train the model to predict the noise, from which we can compute the mean:

$$\boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, \boldsymbol{x}_0) = \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t - \frac{\beta_t}{\sqrt{1-\overline{\alpha}_t}}\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)\right) \tag{25.22}$$

With this parameterization, the loss (averaged over the dataset) becomes

$$L_{t-1} = \mathbb{E}_{\boldsymbol{x}_0 \sim q_0(\boldsymbol{x}_0), \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})}\left[\underbrace{\frac{\beta_t^2}{2\sigma_t^2 \alpha_t (1-\overline{\alpha}_t)}}_{\lambda_t} ||\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_{\boldsymbol{\theta}}\left(\underbrace{\sqrt{\overline{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\overline{\alpha}_t}\boldsymbol{\epsilon}}_{\boldsymbol{x}_t}, t\right)||^2\right] \tag{25.23}$$

The time dependent weight λ_t ensures that the training objective corresponds to maximum likelihood training (assuming the variational bound is tight). However, it has been found empirically that the model produces better looking samples if we set $\lambda_t = 1$. The resulting simplified loss (also averaging over time steps t in the model) is given by

$$L_{\text{simple}} = \mathbb{E}_{\boldsymbol{x}_0 \sim q_0(\boldsymbol{x}_0), \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), t \sim \text{Unif}(1,T)}\left[||\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_{\boldsymbol{\theta}}\left(\underbrace{\sqrt{\overline{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\overline{\alpha}_t}\boldsymbol{\epsilon}}_{\boldsymbol{x}_t}, t\right)||^2\right] \tag{25.24}$$

The overall training procedure is shown in Algorithm 25.1. We can improve the perceptual quality of samples using more advanced weighting schemes, are discussed in [Cho+22]. Conversely, if the goal is to improve likelihood scores, we can optimize the noise schedule, as discussed in Section 25.2.4.

Algorithm 25.1: Training a DDPM model with L_{simple}.

while *not converged* **do**
  $\boldsymbol{x}_0 \sim q_0(\boldsymbol{x}_0)$
  $t \sim \text{Unif}(\{1, \ldots, T\})$
  $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
  Take gradient descent step on $\nabla_{\boldsymbol{\theta}}||\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_{\boldsymbol{\theta}}\left(\sqrt{\overline{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\overline{\alpha}_t}\boldsymbol{\epsilon}, t\right)||^2$

After the model is trained, we can generate data using ancestral sampling, as shown in Algorithm 25.2.

25.2.4 Learning the noise schedule

In this section, we describe a way to optimize the noise schedule used by the encoder so as to maximize the ELBO; this approach is called a **variational diffusion model** or **VDM** [Kin+21].

Algorithm 25.2: Sampling from a DDPM model.

$\boldsymbol{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
foreach $t = T, \ldots, 1$ **do**
  $\boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
  $\boldsymbol{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\boldsymbol{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t) \right) + \sigma_t \boldsymbol{\epsilon}_t$
Return $\boldsymbol{x}_0$

We will use the following parameterization of the encoder:

$$q(\boldsymbol{x}_t|\boldsymbol{x}_0) = \mathcal{N}(\boldsymbol{x}_t|\hat{\alpha}_t \boldsymbol{x}_0, \hat{\sigma}_t^2 \mathbf{I}) \tag{25.25}$$

(Note that $\hat{\alpha}_t$ and $\hat{\sigma}_t$ are different to the parameters α_t and σ_t in Section 25.2.1.) Rather than working with $\hat{\alpha}_t$ and $\hat{\sigma}_t^2$ separately, we will learn to predict their ratio, which is known as the **signal to noise ratio**:

$$R(t) = \hat{\alpha}_t^2 / \hat{\sigma}_t^2 \tag{25.26}$$

This should be monotonically decreasing in t. This can be ensured by defining $R(t) = \exp(-\gamma_{\boldsymbol{\phi}}(t))$, where $\gamma_{\boldsymbol{\phi}}(t)$ is a monotonic neural network. We usually set $\hat{\alpha}_t = \sqrt{1 - \sigma_t^2}$, to corresponde to the variance preserving SDE discussed in Section 25.4.

Following the derivation in Section 25.2.3, the negative ELBO (variational upper bound) can be written as

$$\mathcal{L}(\boldsymbol{x}_0) = \underbrace{D_{\mathbb{KL}}\left(q(\boldsymbol{x}_T|\boldsymbol{x}_0) \parallel p(\boldsymbol{x}_T)\right)}_{\text{prior loss}} + \underbrace{\mathbb{E}_{q(\boldsymbol{x}_1|\boldsymbol{x}_0)}[-\log p_{\boldsymbol{\theta}}(\boldsymbol{x}_0|\boldsymbol{x}_1)]}_{\text{reconstruction loss}} + \underbrace{\mathcal{L}_D(\boldsymbol{x}_0)}_{\text{diffusion loss}} \tag{25.27}$$

where the first two terms are similar to a standard VAE, and the final diffusion loss is given below:[3]

$$\mathcal{L}_D(\boldsymbol{x}_0) = \frac{1}{2} \mathbb{E}_{\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})} \int_0^1 R'(t) \left\| \boldsymbol{x}_0 - \hat{\boldsymbol{x}}_{\boldsymbol{\theta}}(\boldsymbol{z}_t, t) \right\|_2^2 dt \tag{25.28}$$

where $R'(t)$ is the derivative of the SNR function, and $\boldsymbol{z}_t = \alpha_t \boldsymbol{x}_0 + \sigma_t \boldsymbol{\epsilon}_t$. (See [Kin+21] for the derivation.)

Since the SNR functon is invertible, due to the monotonicity assumtion, we can perform a change of variables, and make everything a function of $v = R(t)$ instead of t. In particular, let $\boldsymbol{z}_v = \alpha_v \boldsymbol{x}_0 + \sigma_v \boldsymbol{\epsilon}$, and $\tilde{\boldsymbol{x}}_{\boldsymbol{\theta}}(\boldsymbol{z}, v) = \hat{\boldsymbol{x}}_{\boldsymbol{\theta}}(\boldsymbol{z}, R^{-1}(v))$. Then we can rewrite Equation (25.28) as

$$\mathcal{L}_D(\boldsymbol{x}_0) = \frac{1}{2} \mathbb{E}_{\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})} \int_{R_{\min}}^{R_{\max}} \left\| \boldsymbol{x}_0 - \tilde{\boldsymbol{x}}_{\boldsymbol{\theta}}(\boldsymbol{z}_v, v) \right\|_2^2 dv \tag{25.29}$$

where $R_{\min} = R(1)$ and $R_{\max} = R(0)$. Thus we see that the shape of the SNR schedule does not matter, except for its value at the two end points.

3. We present a simplified form of the loss that uses the continuous time limit, which we discuss in Section 25.4.

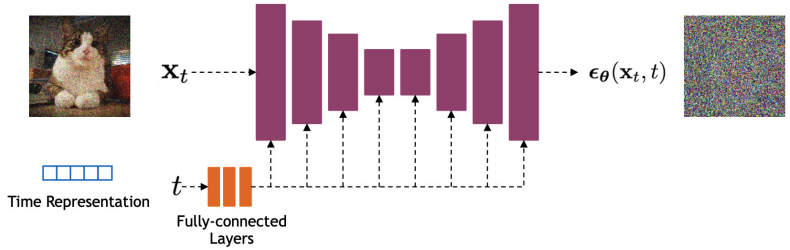


Figure 25.3: Illustration of the U-net architecture used in the denoising step. From Slide 26 of [KGV22]. Used with kind permission of Arash Vahdat.

(a) *(b)* *(c)*

Figure 25.4: Some sample images generated by a small variational diffusion model trained on EMNIST for about 30 minutes on a K40 GPU. (a) Unconditional sampling. (b) Conditioned on class label. (c) Using classifier-free guidance (see Section 25.6.3). Generated by diffusion_ emnist.ipynb. Used with kind permission of Alex Alemi.

The integral in Equation (25.29) can be estimated by sampling a timestep uniformly at random. When processing a minibatch of k examples, we can produce a lower variance estimate of the variational bound by using a **low-discrepancy sampler** (cf., Section 11.6.5). In this approach, instead of sampling the timesteps independently, we sample a single uniform random number $u_0 \sim \text{Unif}(0, 1)$, and then set $t^i = \mod (u_0 + i/k, 1)$ for the i'th sample. We can also optimize the noise schedule wrt the variance of the diffusion loss.

25.2.5 Example: image generation

Diffusion models are often used to generate images. The most common architecture for image generation is based on the **U-net** model [RFB15], as shown in Figure 25.3. The time step t is encoded as a vector, using sinusoidal positional encoding or random Fourier features, and is then fed into the residual blocks, using either simple spatial addition or by conditioning the group norm layers [DN21a]. Of course, other architectures besides U-net are possible. For example, recently [PX22; Li+22; Bao+22a] have proposed the use of transformers, to replace the convolutional and

deconvolutional layers.

The results of training a small U-net VDM on EMNIST images are shown in Figure 25.4. By training big models (billions of parameters) for a long time (days) on lots of data (millions of images), diffusion models can be made to generate very high quality images (see Figure 20.2). Results can be further improved by using conditional diffusion models, where guidance is provided about what kinds of images to generate (see Section 25.6).

25.3 Score-based generative models (SGMs)

This section is written with Yang Song and Durk Kingma.

In Section 24.3, we discussed how to fit energy based models (EBMs) using score matching. This adjusts the parameters of the EBM so that the score function of the model, $\nabla_{\boldsymbol{x}} \log p_{\boldsymbol{\theta}}(\boldsymbol{x})$, matches the score function of the data, $\nabla_{\boldsymbol{x}} \log p_{\mathcal{D}}(\boldsymbol{x})$. An alternative to fitting a scalar energy function and computing its score is to directly learn the score function. This is called a **score-based generative model** or **SGM** [SE19; SE20b; Son+21b]. We can optimize the score function $\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x})$ using basic score matching (Section 24.3.1), sliced score matching (Section 24.3.3 or denoising score matching (Section 24.3.2). We discuss this class of models in more detail below. (For a comparison with EBMs, see [SH21].)

25.3.1 Example

In Figure 25.5a, we show the **Swiss roll** dataset. We estimate the score function by fitting an MLP with 2 hidden layers, each with 128 hidden units, using basic score matching. In Figure 25.5b, we show the output of the network after training for 10,000 steps of SGD. We see that there are no major false negatives (since wherever the density of the data is highest, the gradient field is zero), but there are some false positives (since some regions of zero gradient do not correspond to data regions). The comparison of the predicted outputs with the empirical data density is shown more clearly in Figure 25.5c. In Figure 25.5d, we show some samples from the learned model, generated using Langevin sampling.

25.3.2 Adding noise at multiple scales

In general, score matching can have difficulty when there are regions of low data density. To see this, suppose $p_{\mathcal{D}}(\mathbf{x}) = \pi p_0(\mathbf{x}) + (1-\pi)p_1(\mathbf{x})$. Let $\mathcal{S}_0 := \{\mathbf{x} \mid p_0(\mathbf{x}) > 0\}$ and $\mathcal{S}_1 := \{\mathbf{x} \mid p_1(\mathbf{x}) > 0\}$ be the supports of $p_0(\mathbf{x})$ and $p_1(\mathbf{x})$ respectively. When they are disjoint from each other, the score of $p_{\mathcal{D}}(\mathbf{x})$ is given by

$$\nabla_{\mathbf{x}} \log p_{\mathcal{D}}(\mathbf{x}) = \begin{cases} \nabla_{\mathbf{x}} \log p_0(\mathbf{x}), & \mathbf{x} \in \mathcal{S}_0 \\ \nabla_{\mathbf{x}} \log p_1(\mathbf{x}), & \mathbf{x} \in \mathcal{S}_1, \end{cases} \tag{25.30}$$

which does not depend on the weight π. Hence score matching cannot correctly recover the true distribution. Furthermore, Langevin sampling will have difficulty traversing between modes. (In practice this will happen even when the different modes only have approximately disjoint supports.)

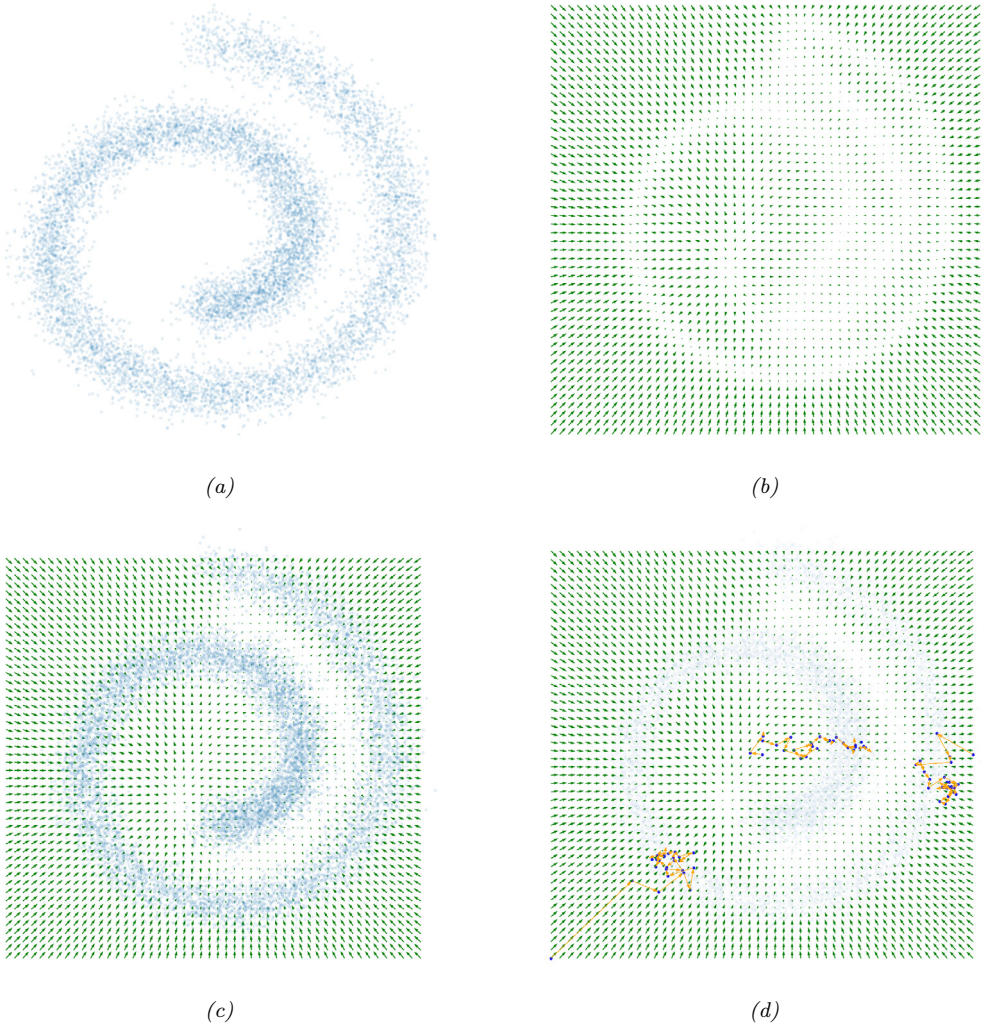


Figure 25.5: Fitting a score-based generative model to the 2d Swiss roll dataset. (a) Training set. (b) Learned score function trained using the basic score matching. (c) Superposition of learned score function and empirical density. (d) Langevin sampling applied to the learned model. We show 3 different trajectories, each of length 25. Generated by score_matching_swiss_roll.ipynb.

Song and Ermon [SE19; SE20b] and Song et al. [Son+21b] overcome this difficulty by perturbing training data with different scales of noise. Specifically, they use

$$q_\sigma(\tilde{\boldsymbol{x}}|\boldsymbol{x}) = \mathcal{N}(\tilde{\boldsymbol{x}}|\boldsymbol{x}, \sigma^2\mathbf{I}) \tag{25.31}$$

$$q_\sigma(\tilde{\boldsymbol{x}}) = \int p_{\mathcal{D}}(\boldsymbol{x}) q_\sigma(\tilde{\boldsymbol{x}}|\boldsymbol{x}) d\boldsymbol{x} \tag{25.32}$$

For a large noise perturbation, different modes are connected due to added noise, and the estimated weights between them are therefore accurate. For a small noise perturbation, different modes are more disconnected, but the noise-perturbed distribution is closer to the original unperturbed data distribution. Using a sampling method such as annealed Langevin dynamics [SE19; SE20b; Son+21b] or diffusion sampling [SD+15a; HJA20; Son+21b], we can sample from the most noise-perturbed distribution first, then smoothly reduce the magnitude of noise scales until reaching the smallest one. This procedure helps combine information from all noise scales, and maintains the correct estimation of weights from higher noise perturbations when sampling from smaller ones.

In practice, all score models share weights and are implemented with a single neural network conditioned on the noise scale; this is called a **noise conditional score network**, and has the form $\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x}, \sigma)$. Scores of different scales are estimated by training a mixture of score matching objectives, one per noise scale. If we use the denoising score matching objective in Equation (24.33), we get

$$\mathcal{L}(\boldsymbol{\theta}; \sigma) = \mathbb{E}_{q(\mathbf{x},\tilde{\mathbf{x}})}\left[\frac{1}{2}\left\|\nabla_{\mathbf{x}} \log p_{\boldsymbol{\theta}}(\tilde{\mathbf{x}}, \sigma) - \nabla_{\mathbf{x}} \log q_\sigma(\tilde{\mathbf{x}}|\mathbf{x})\right\|_2^2\right] \tag{25.33}$$

$$= \frac{1}{2}\mathbb{E}_{p_{\mathcal{D}}(\mathbf{x})}\mathbb{E}_{\tilde{\mathbf{x}}\sim\mathcal{N}(\boldsymbol{x},\sigma^2\mathbf{I})}\left\{\left\|\boldsymbol{s}_{\boldsymbol{\theta}}(\tilde{\mathbf{x}}, \sigma) + \frac{(\tilde{\mathbf{x}} - \boldsymbol{x})}{\sigma^2}\right\|_2^2\right\} \tag{25.34}$$

where we used the fact that, for a Gaussian, the score is given by

$$\nabla_{\boldsymbol{x}} \log \mathcal{N}(\tilde{\boldsymbol{x}}|\boldsymbol{x}, \sigma^2\mathbf{I}) = -\nabla_{\boldsymbol{x}} \frac{1}{2\sigma^2}(\boldsymbol{x} - \tilde{\boldsymbol{x}})^\mathsf{T}(\boldsymbol{x} - \tilde{\boldsymbol{x}}) = \frac{\boldsymbol{x} - \tilde{\boldsymbol{x}}}{\sigma^2} \tag{25.35}$$

If we have T different noise scales, we can combine the losses in a weighted fashion using

$$\mathcal{L}(\boldsymbol{\theta}; \sigma_{1:T}) = \sum_{t=1}^{T} \lambda_t \mathcal{L}(\boldsymbol{\theta}; \sigma_t) \tag{25.36}$$

where we choose $\sigma_1 > \sigma_2 > \cdots > \sigma_T$, and the weighting term satisfies $\lambda_t > 0$.

25.3.3 Equivalence to DDPM

We now show that the above score-based generative model training objective is equivalent to the DDPM loss. To see this, first let us replace $p_{\mathcal{D}}(\boldsymbol{x})$ with $q_0(\boldsymbol{x}_0)$, $\tilde{\boldsymbol{x}}$ with $\boldsymbol{x}_t$, and $\boldsymbol{s}_{\boldsymbol{\theta}}(\tilde{\boldsymbol{x}}, \sigma)$ with $\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)$. We will also compute a stochastic approximation to Equation (25.36) by sampling a time step uniformly at random. Then Equation (25.36) becomes

$$\mathcal{L} = \mathbb{E}_{\boldsymbol{x}_0\sim q_0(\boldsymbol{x}_0), \boldsymbol{x}_t\sim q(\boldsymbol{x}_t|\boldsymbol{x}_0), t\sim\text{Unif}(1,T)}\left[\lambda_t \left\|\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t) + \frac{(\boldsymbol{x}_t - \boldsymbol{x}_0)}{\sigma_t^2}\right\|_2^2\right] \tag{25.37}$$

If we use the fact that $\boldsymbol{x}_t = \boldsymbol{x}_0 + \sigma_t \boldsymbol{\epsilon}$, and if we define $\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t) = -\frac{\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)}{\sigma_t}$, we can rewrite this as

$$\mathcal{L} = \mathbb{E}_{\boldsymbol{x}_0 \sim q_0(\boldsymbol{x}_0), \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), t \sim \text{Unif}(1,T)} \left[\frac{\lambda_t}{\sigma_t^2} \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t) \right\|_2^2 \right] \tag{25.38}$$

If we set $\lambda_t = \sigma_t^2$, we recover L_{simple} loss in Equation (25.24).

25.4 Continuous time models using differential equations

In this section, we consider a DDPM model in the limit of an infinite number of hidden layers, or equivalently, an SGM in the limit of an infinite number of noise levels. This requires switching from discrete time to continuous time, which complicates the mathematics. The advantage is that we can leverage the large existing literature on solvers for ordinary and stochastic differential equations to enable faster generation, as we will see.

25.4.1 Forwards diffusion SDE

Let us first consider a diffusion process where the noise level β_t gets rewritten as $\beta(t)\Delta t$, where Δt is a step size:

$$\boldsymbol{x}_t = \sqrt{1-\beta_t}\boldsymbol{x}_{t-1} + \sqrt{\beta_t}\mathcal{N}(\mathbf{0}, \mathbf{I}) = \sqrt{1-\beta(t)\Delta t}\boldsymbol{x}_{t-1} + \sqrt{\beta(t)\Delta t}\mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{25.39}$$

If Δt is small, we can approximate the first term with a first-order Taylor series expansion to get

$$\boldsymbol{x}_t \approx \boldsymbol{x}_{t-1} - \frac{\beta(t)\Delta t}{2}\boldsymbol{x}_{t-1} + \sqrt{\beta(t)\Delta t}\mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{25.40}$$

Hence for small Δt we have

$$\frac{\boldsymbol{x}_t - \boldsymbol{x}_{t-1}}{\Delta t} \approx -\frac{\beta(t)}{2}\boldsymbol{x}_{t-1} + \frac{\sqrt{\beta(t)}}{\sqrt{\Delta t}}\mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{25.41}$$

We can now switch to the **continuous time** limit, and write this as the following **stochastic differential equation** or **SDE**:

$$\frac{d\boldsymbol{x}(t)}{dt} = -\frac{1}{2}\beta(t)\boldsymbol{x}(t) + \sqrt{\beta(t)}\frac{d\boldsymbol{w}(t)}{dt} \tag{25.42}$$

where $\boldsymbol{w}(t)$ represents a standard **Wiener process**, also called **Brownian noise**. More generally, we can write such SDEs as follows, where we use **Itô calculus** notation (see e.g., [SS19]):

$$d\boldsymbol{x} = \underbrace{\boldsymbol{f}(\boldsymbol{x}, t)}_{\text{drift}} dt + \underbrace{g(t)}_{\text{diffusion}} d\boldsymbol{w} \tag{25.43}$$

The first term in the above SDE is called the **drift coefficient**, and the second term is called the **diffusion coefficient**.

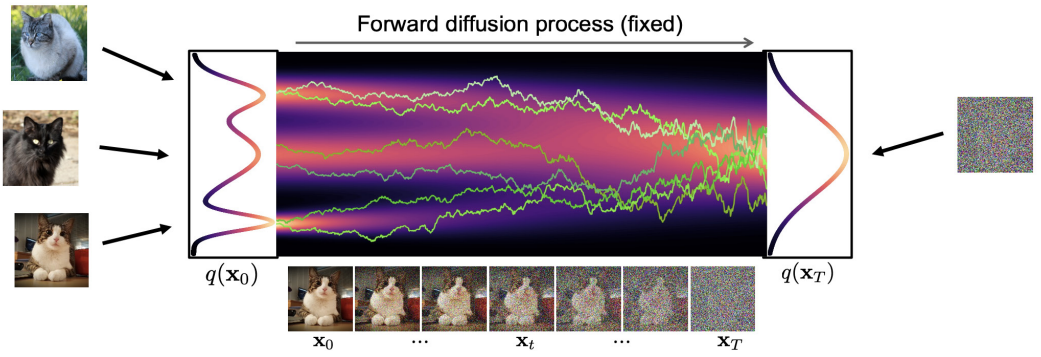


Figure 25.6: Illustration of the forwards diffusion process in continuous time. Yellow lines are sample paths from the SDE. Heat map represents the marginal distribution computed using the probability flow ODE. From Slide 43 of [KGV22]. Used with kind permission of Karsten Kreis.

We can gain some intuition for these processes by looking at the 1d example in Figure 25.6. We can draw multiple paths as follows: sample an initial state from the data distribution, and then integrate over time using **Euler-Maruyama** integration:

$$\boldsymbol{x}(t+\Delta t) = \boldsymbol{x}(t) + \boldsymbol{f}(\boldsymbol{x}(t), t)\Delta t + g(t)\sqrt{\Delta t}\mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{25.44}$$

We can see how the data distribution at $t = 0$, on the left hand side, gradually gets transformed to a pure noise distribution at $t = 1$, on the right hand side.

In [Son+21b], they show that the SDE corresponding to DDPMs, in the $T \to \infty$ limit, is given by

$$d\boldsymbol{x} = -\frac{1}{2}\beta(t)\boldsymbol{x}dt + \sqrt{\beta(t)}d\boldsymbol{\omega} \tag{25.45}$$

where $\beta(t/T) = T\beta_t$. Here the drift term is proportional to $-\boldsymbol{x}$, which encourages the process to return to 0. Consequently, DDPM corresponds to a **variance preserving** process. By contrast, the SDE corresponding to SGMs is given by the following:

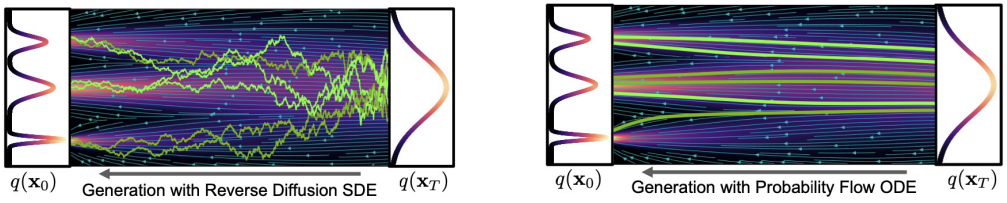
$$d\boldsymbol{x} = \sqrt{\frac{d[\sigma(t)^2]}{dt}}d\boldsymbol{\omega} \tag{25.46}$$

where $\sigma(t/T) = \sigma_t$. This SDE has zero drift, so corresponds to a **variance exploding** process.

25.4.2 Forwards diffusion ODE

Instead of adding Gaussian noise at every step, we can just sample the initial state, and then let it evolve deterministically over time according to the following **ordinary differential equation** or **ODE**:

$$d\boldsymbol{x} = \underbrace{\left[f(\boldsymbol{x}, t) - \frac{1}{2}g(t)^2\nabla_{\boldsymbol{x}} \log p_t(\boldsymbol{x})\right]}_{h(\boldsymbol{x},t)} dt \tag{25.47}$$



(a) *(b)*

Figure 25.7: Illustration of the reverse diffusion process. (a) Sample paths from the SDE. (b) Deterministic trajectories from the probability flow ODE. From Slide 65 of [KGV22]. Used with kind permission of Karsten Kreis.

This is called the **probability flow ODE** [Son+21b, Sec D.3]. We can compute the state at any moment in time using any ODE solver:

$$\boldsymbol{x}(t) = \boldsymbol{x}(0) + \int_0^t h(\boldsymbol{x}, \tau) d\tau \tag{25.48}$$

See Figure 25.7b for a visualization of a sample trajectory. If we start the solver from different random states $\boldsymbol{x}(0)$, then the induced distribution over paths will have the same marginals as the SDE model. See the heatmap in Figure 25.6 for an illustration.

25.4.3 Reverse diffusion SDE

To generate samples from this model, we need to be able to reverse the SDE. In a remarkable result, [And82] showed that any forwards SDE of the form in Equation (25.43) can be reversed to get the following **reverse-time SDE**:

$$d\boldsymbol{x} = \left[f(\boldsymbol{x}, t) - g(t)^2 \nabla_{\boldsymbol{x}} \log q_t(\boldsymbol{x})\right] dt + g(t) d\overline{\boldsymbol{w}} \tag{25.49}$$

where $\overline{\boldsymbol{w}}$ is the standard Wiener process when time flows backwards, dt is an infinitesimal negative time step, and $\nabla_{\boldsymbol{x}} \log q_t(\boldsymbol{x})$ is the score function.

In the case of the DDPM, the reverse SDE has the following form:

$$d\boldsymbol{x}_t = \left[-\frac{1}{2}\beta(t)\boldsymbol{x}_t - \beta(t)\nabla_{\boldsymbol{x}_t} \log q_t(\boldsymbol{x}_t)\right] dt + \sqrt{\beta(t)} d\overline{\boldsymbol{w}}_t \tag{25.50}$$

To estimate the score function, we can use denoising score matching as we discussed in Section 25.3, to get

$$\nabla_{\boldsymbol{x}_t} \log q_t(\boldsymbol{x}_t) \approx \boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t) \tag{25.51}$$

(In practice, it is advisable to use variance reduction techniques, such as importance sampling, as discussed in [Son+21a].) The SDE becomes

$$d\boldsymbol{x}_t = -\frac{1}{2}\beta(t)\left[\boldsymbol{x}_t + 2\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)\right] dt + \sqrt{\beta(t)} d\overline{\boldsymbol{w}}_t \tag{25.52}$$

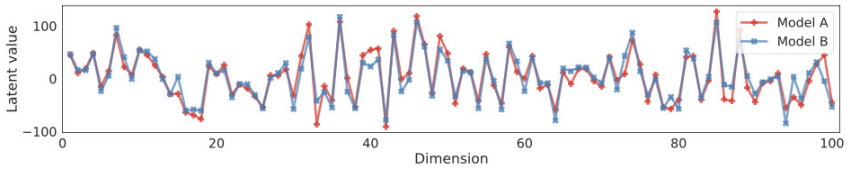


Figure 25.8: Comparing the first 100 dimensions of the latent code obtained for a random CIFAR-100 image. "Model A" and "Model B" are separately trained with different architectures. From Figure 7 of [Son+21b]. Used with kind permission of Yang Song.

After fitting the score network, we can sample from it using ancestral sampling (as in Section 25.2), or we can use the Euler-Maruyama integration scheme in Equation (25.44), which gives

$$\boldsymbol{x}_{t-1} = \boldsymbol{x}_t + \frac{1}{2}\beta(t)\left[\boldsymbol{x}_t + 2\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)\right]\Delta t + \sqrt{\beta(t)\Delta t}\mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{25.53}$$

See Figure 25.7a for an illustration.

25.4.4 Reverse diffusion ODE

Based on the results in Section 25.4.2, we can derive the probability flow ODE from the reverse-time SDE in Equation (25.49) to get

$$d\boldsymbol{x}_t = \left[f(\boldsymbol{x}, t) - \frac{1}{2}g(t)^2\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)\right] dt \tag{25.54}$$

If we set $f(\boldsymbol{x}, t) = -\frac{1}{2}\beta(t)$ and $g(t) = \sqrt{\beta(t)}$, as in DDPM, this becomes

$$d\boldsymbol{x}_t = -\frac{1}{2}\beta(t)\left[\boldsymbol{x}_t + \boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)\right] dt \tag{25.55}$$

See Figure 25.7b for an illustration. A simple way to solve this ODE is to use **Euler's method**:

$$\boldsymbol{x}_{t-1} = \boldsymbol{x}_t + \frac{1}{2}\beta(t)\left[\boldsymbol{x}_t + \boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)\right]\Delta t \tag{25.56}$$

However, in practice one can get better results using higher-order ODE solvers, such as **Heun's method** [Kar+22].

This model is a special case of a **neural ODE**, also called a **continuous normalizing flow** (see Section 23.2.6). Consequently we can derive the exact log marginal likelihood. However, instead of maximizing this directly (which is expensive), we use score matching to fit the model.

Another advantage of the deterministic ODE approach is that it guarantees that the generative model is **identifiable**. To see this, note that the ODE (in both forwards and reverse directions) is deterministic, and is uniquely determined by the score function. If the architecture is sufficiently flexible, and if there is enough data, then score matching will recover the true score function of the data generating process. Thus, after training, a given datapoint will map to a unique point in latent space, regardless of the model architecture or initialization (see Figure 25.8).

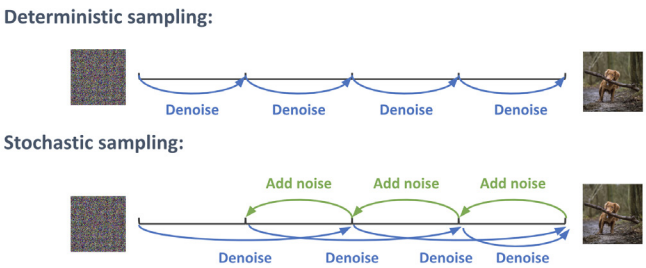


Figure 25.9: Generatng from the reverse diffusion process using 4 steps. (Top) Deterministic sampling. (Bottom) A mix of deterministic and stochastic sampling. Used with kind permission of Ruiqi Gao.

Furthermore, since every point in latent space decodes to a unique image, we can perform "semantic interpolation" in the latent space to generate images with properties that are in between two input examples (cf., Figure 20.9).

25.4.5 Comparison of the SDE and ODE approach

In Section 25.4.3 we described the reverse diffusion process as an SDE, and in Section 25.4.4, we described it as an ODE. We can see the connection between these methods by rewriting the SDE in Equation (25.49) as follows:

$$d\boldsymbol{x}_t = \underbrace{-\frac{1}{2}\beta(t)[\boldsymbol{x}_t + \boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)]dt}_{\text{probability flow ODE}} \underbrace{-\frac{1}{2}\beta(t)\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)dt + \sqrt{\beta(t)}d\overline{\boldsymbol{w}}_t}_{\text{Langevin diffusion SDE}} \tag{25.57}$$

The continuous noise injection can compensate for errors introduced by the numerical integration of the ODE term. Consequently, the resulting samples often look better. However, the ODE approach can be faster. Fortunately it is possible to combine these techniques, as proposed in [Kar+22]. The basic idea is illustrated in Figure 25.9: we alternate between performing a deterministic step using an ODE solver, and then adding a small amount noise to the result. This can be repeated for some number of steps. (We discuss ways to reduce the number of required steps in Section 25.5.)

25.4.6 Example

A simple JAX implementation of the above ideas, written by Winnie Xu, can be found in diffusion_mnist.ipynb. This fits a small model to MNIST images using denoising score matching. It then generates from the model by solving the probability flow ODE using the diffrax library. By scaling this kind of method up to a much larger model, and training for a much longer time, it is possible to produce very impressive looking results, as shown in Figure 25.10.

25.5 Speeding up diffusion models

One of the main disadvantages of diffusion models is that generating from them takes many small steps, which can be slow. While it is possible to just take fewer, larger steps, the results are much

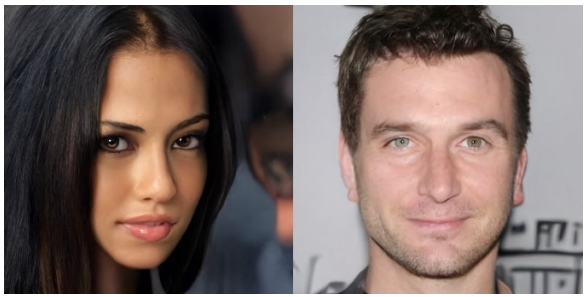


Figure 25.10: Synthetic faces from a score-based generative model trained on CelebA-HQ-256 images. From Figure 12 of [Son+21b]. Used with kind permission of Yang Song.

worse. In this section, we briefly mention a few techniques that have been proposed to tackle this important problem. Many other techniques are mentioned in the recent review papers [UAP22; Yan+22; Cao+22].

25.5.1 DDIM sampler

In this section, we describe the denoising diffusion implicit model or **DDIM** of [SME21], which can be used for efficient deterministic generation. The first step is to use a **non-Markovian** forwards diffusion process, so it always conditions on the input in addition to the previous step:

$$q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0) = \mathcal{N}(\sqrt{\overline{\alpha}_{t-1}}\boldsymbol{x}_0 + \sqrt{1-\overline{\alpha}_{t-1}-\tilde{\sigma}_t^2}\frac{\boldsymbol{x}_t - \sqrt{\overline{\alpha}_t}\boldsymbol{x}_0}{\sqrt{1-\overline{\alpha}_t}}, \tilde{\sigma}_t^2\mathbf{I}) \tag{25.58}$$

The corresponding reverse process is

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) = \mathcal{N}(\sqrt{\overline{\alpha}_{t-1}}\hat{\boldsymbol{x}}_0 + \sqrt{1-\overline{\alpha}_{t-1}-\tilde{\sigma}_t^2}\frac{\boldsymbol{x}_t - \sqrt{\overline{\alpha}_t}\hat{\boldsymbol{x}}_0}{\sqrt{1-\overline{\alpha}_t}}, \tilde{\sigma}_t^2\mathbf{I}) \tag{25.59}$$

where $\hat{\boldsymbol{x}}_0 = \hat{\boldsymbol{x}}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)$ is the predicted output from the model. By setting $\tilde{\sigma}_t^2 = 0$, the reverse process becomes fully deterministic, given the initial prior sample (whose variance is controlled by $\tilde{\sigma}_T^2$). The resulting probability flow ODE gives better results when using a small number of steps compared to the methods discussed in Section 25.4.4.

Note that the weighted negative VLB for this model is the same as L_{simple} in Section 25.2, so the DDIM sampler can be applied to a trained DDPM model.

25.5.2 Non-Gaussian decoder networks

If the reverse diffusion process takes larger steps, then the induced distribution over clean outputs given a noisy input will become multimodal, as illustrated in Figure 25.11. This requires more complicated forms for the distribution $p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$. In [Gao+21], they use an EBM to fit this distribution. However, this still requires the use of MCMC to draw a sample. In [XKV22], they use a GAN (Chapter 26) to fit this distribution. This enables us to easily draw a sample by passing

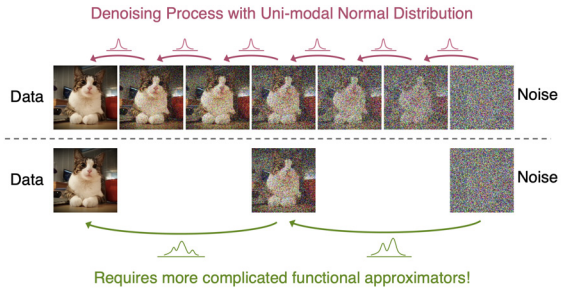


Figure 25.11: Illustration of why taking larger steps in the reverse diffusion process needs more complex, mulit-modal conditional distributions. From Slide 90 of [KGV22]. Used with kind permission of Arash Vahdat.

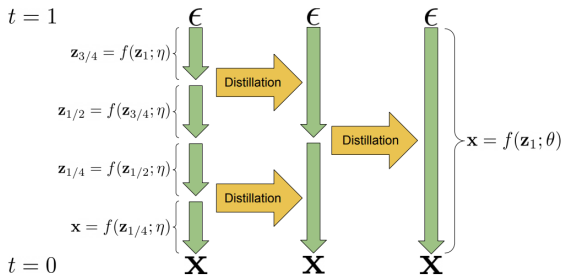


Figure 25.12: Progressive distillation. From Figure 1 of [SH22]. Used with kind permission of Tim Salimans.

Gaussian noise through the generator. The benefits over a single stage GAN is that both the generator and discriminator are solving a much simpler problem, resulting in increased mode coverage, and better training stability. The benefit over a standard diffusion model is that we can generate high quality samples in many fewer steps.

25.5.3 Distillation

In this section, we discuss the **progressive distillation** method of [SH22], which provides a way to create a diffusion model that only needs a small number of steps to create high quality samples. The basic idea is follows. First we train a DDPM model in the usual way, and sample from it using the DDIM method; we treat this as the "teacher" model. We use this to generate intermediate latent states, and train a "student" model to predict the output of the teacher on every second step, as shown in Figure 25.12. After the student has been trained, it can generate results that are as good as the teacher, but in half the number of steps. This student can then teach a new generation of even faster students. See Algorithm 25.4 for the pseudocode, which should be compared to Algorithm 25.3 for the standard training procedure. Note that each round of teaching becomes faster, because the teachers become smaller, so the total time to perform the distillation is relatively small. The resulting model can generate high quality samples in as few as 4 steps.

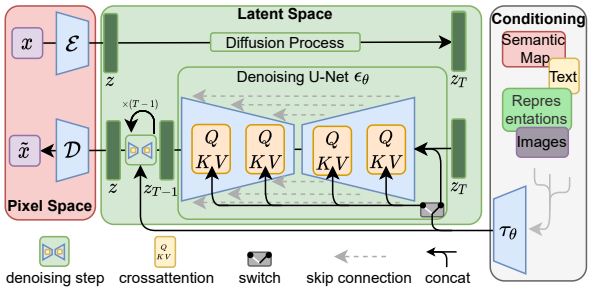


Figure 25.13: Combining a VAE with a diffusion model. Here $\mathcal{E}$ and $\mathcal{D}$ are the encoder and decoder of the VAE. The diffusion model conditions on the inputs either by using concatentation or by using a cross-attention mechanism. From Figure 3 of [Rom+22]. Used with kind permission of Robin Rombach.

Algorithm 25.3: Standard training

Input: Model $\hat{\boldsymbol{x}}_{\boldsymbol{\theta}}(\boldsymbol{z}_t)$ to be trained
Input: Dataset $\mathcal{D}$
Input: Loss weight function w

while *not converged* **do**
  $\boldsymbol{x} \sim \mathcal{D}$
  $t \sim \text{Unif}(0, 1)$
  $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
  $\boldsymbol{z}_t = \alpha_t \boldsymbol{x} + \sigma_t \boldsymbol{\epsilon}$
  $\tilde{\boldsymbol{x}} = \boldsymbol{x}$ (Clean data is target)
  $\lambda_t = \log(\alpha_t^2/\sigma_t^2)$ (Log SNR)
  $L_{\boldsymbol{\theta}} = w(\lambda_t) \|\tilde{\boldsymbol{x}} - \hat{\boldsymbol{x}}_{\boldsymbol{\theta}}(\boldsymbol{z}_t)\|_2^2$
  $\boldsymbol{\theta} := \boldsymbol{\theta} - \gamma \nabla_{\boldsymbol{\theta}} L_{\boldsymbol{\theta}}$

Algorithm 25.4: Progressive distillation

Input: Trained teacher model $\hat{\boldsymbol{x}}_{\boldsymbol{\eta}}(\boldsymbol{z}_t)$
Input: Dataset $\mathcal{D}$
Input: Loss weight function w
Input: Student sampling steps N
foreach *K iterations* **do**
  $\boldsymbol{\theta} := \boldsymbol{\eta}$ (Assign student)
  while *not converged* **do**
    $\boldsymbol{x} \sim \mathcal{D}$
    $t = i/N,\ i \sim \text{Cat}(1, 2, \ldots, N)$
    $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
    $\boldsymbol{z}_t = \alpha_t \boldsymbol{x} + \sigma_t \boldsymbol{\epsilon}$
    $t' = t - 0.5/N,\ t'' = t - 1/N$
    $\boldsymbol{z}_{t'} = \alpha_{t'} \hat{\boldsymbol{x}}_{\boldsymbol{\eta}}(\boldsymbol{z}_t) + \frac{\sigma_{t'}}{\sigma_t}(\boldsymbol{z}_t - \alpha_t \hat{\boldsymbol{x}}_{\boldsymbol{\eta}}(\boldsymbol{z}_t))$
    $\boldsymbol{z}_{t''} = \alpha_{t''} \hat{\boldsymbol{x}}_{\boldsymbol{\eta}}(\boldsymbol{z}_{t'}) + \frac{\sigma_{t''}}{\sigma_{t'}}(\boldsymbol{z}_{t'} - \alpha_{t'} \hat{\boldsymbol{x}}_{\boldsymbol{\eta}}(\boldsymbol{z}_{t'}))$
    $\tilde{\boldsymbol{x}} = \frac{\boldsymbol{z}_{t''} - (\sigma_{t''}/\sigma_t)\boldsymbol{z}_t}{\alpha_{t''} - (\sigma_{t''}/\sigma_t)\alpha_t}$ (Teacher is target)
    $\lambda_t = \log(\alpha_t^2/\sigma_t^2)$
    $L_{\boldsymbol{\theta}} = w(\lambda_t) \|\tilde{\boldsymbol{x}} - \hat{\boldsymbol{x}}_{\boldsymbol{\theta}}(\boldsymbol{z}_t)\|_2^2$
    $\boldsymbol{\theta} := \boldsymbol{\theta} - \gamma \nabla_{\boldsymbol{\theta}} L_{\boldsymbol{\theta}}$
  $\boldsymbol{\eta} := \boldsymbol{\theta}$ (Student becomes next teacher)
  $N := N/2$ (Halve number of sampling steps)

25.5.4 Latent space diffusion

Another approach to speeding up diffusion models for images is to first embed the images into a lower dimensional space, and then fit the diffusion model to the embeddings. This idea has been pursued in several papers.

In the latent diffusion model (**LDM**) of [Rom+22], they adopt a two-stage training scheme, in which they first fit the VAE, augmented with a perceptual loss, and then fit the diffusion model to

the embedding. The architecture is illustrated in Figure 25.13. The LDM forms the foundation of the very popular **stable diffusion** system created by **Stability AI**. In the latent score-based generative model (**LSGM**) of [VKK21], they first train a hierarchical VAE, and then jointly train the VAE and a diffusion model.

In addition to speed, an additional advantage of combining diffusion models with autoencoders is that it makes it simple to apply diffusion to many different kinds of data, such as text and graphs: we just need to define a suitable architecture to embed the input domain into a continuous space. Note, however, that it is also possible to define diffusion directly on discrete state spaces, as we discuss in Section 25.7.

So far we have discussed applying diffusion "on top of" a VAE. However, we can also do the reverse, and fit a VAE on top of a DDPM model, where we use the diffusion model to "post process" blurry samples coming from the VAE. See [Pan+22] for details.

25.6 Conditional generation

In this section, we discuss how to generate samples from a diffusion model where we condition on some side information $\boldsymbol{c}$, such as a class label or text prompt.

25.6.1 Conditional diffusion model

The simplest way to control the generation from a generative model is to train it on $(\boldsymbol{c}, \boldsymbol{x})$ pairs so as to maximize the conditional likelihood, $p(\boldsymbol{x}|\boldsymbol{c})$. If the conditioning signal c is a scalar (e.g., a class label), it can be mapped to an embedding vector, and then incorporated into the network by spatial addition, or by using it to modular the group normalization layers. If the input $\boldsymbol{c}$ is another image, we can simply concatenate it with $\boldsymbol{x}_t$ as an extra set of channels. If the input $\boldsymbol{c}$ is a text prompt, we can embed it, and then use spatial addition or cross-attention (see Figure 25.13 for an illustration).

25.6.2 Classifier guidance

One problem with conditional diffusion models is that we need to retrain them for each kind of conditioning that we want to perform. An alternative approach, known as **classifier guidance** was proposed in [DN21b], and allows us to leverage pre-trained discriminative classifiers of the form $p_{\boldsymbol{\phi}}(\boldsymbol{c}|\boldsymbol{x})$ to control the generation process. The idea is as follows. First we use Bayes' rule to write

$$\log p(\boldsymbol{x}|\boldsymbol{c}) = \log p(\boldsymbol{c}|\boldsymbol{x}) + \log p(\boldsymbol{x}) - \log p(\boldsymbol{c}) \tag{25.60}$$

from which the score function becomes

$$\nabla_{\boldsymbol{x}} \log p(\boldsymbol{x}|\boldsymbol{c}) = \nabla_{\boldsymbol{x}} \log p(\boldsymbol{x}) + \nabla_{\boldsymbol{x}} \log p(\boldsymbol{c}|\boldsymbol{x}) \tag{25.61}$$

We can now use this conditional score to generate samples, rather than the unconditional score. We can further amplify the influence of the conditioning signal by scaling it by a factor $w > 1$:

$$\nabla_{\boldsymbol{x}} \log p_w(\boldsymbol{x}|\boldsymbol{c}) = \nabla_{\boldsymbol{x}} \log p(\boldsymbol{x}) + w \nabla_{\boldsymbol{x}} \log p(\boldsymbol{c}|\boldsymbol{x}) \tag{25.62}$$

In practice, this can be achieved as follows by generating samples from

$$\boldsymbol{x}_{t-1} \sim \mathcal{N}(\boldsymbol{\mu} + w\boldsymbol{\Sigma}\boldsymbol{g}, \boldsymbol{\Sigma}),\ \boldsymbol{\mu} = \boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t),\ \boldsymbol{\Sigma} = \boldsymbol{\Sigma}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t),\ \boldsymbol{g} = \nabla_{\boldsymbol{x}_t} \log p_{\boldsymbol{\phi}}(\boldsymbol{c}|\boldsymbol{x}_t) \tag{25.63}$$

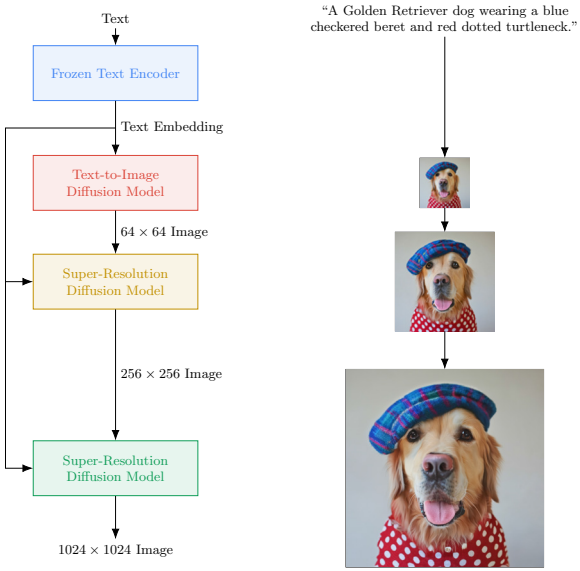


Figure 25.14: Cascaded diffusion model used by the Imagen text-to-image system. From Figure A.5 of [Sah+22b]. Used with kind permission of Saurabh Saxena.

25.6.3 Classifier-free guidance

Unfortunately, $p(\boldsymbol{c}|\boldsymbol{x}_t)$ is a discriminative model, that may ignore many details of the input $\boldsymbol{x}_t$. Hence optimizing along the directions specified by $\nabla_{\boldsymbol{x}_t} \log p(\boldsymbol{c}|\boldsymbol{x}_t)$ can give poor results, similar to what happens when we create adversarial images. In addition, we need to train a classifier for each time step, since $\boldsymbol{x}_t$ will differ in its blurriness.

In [HS21], they proposed a technique called **classifier-free guidance**, which derives the classifier from the generative model, using $p(\boldsymbol{c}|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|\boldsymbol{c})p(\boldsymbol{c})}{p(\boldsymbol{x})}$, from which we get

$$\log p(\boldsymbol{c}|\boldsymbol{x}) = \log p(\boldsymbol{x}|\boldsymbol{c}) + \log p(\boldsymbol{c}) - \log p(\boldsymbol{x}) \tag{25.64}$$

This requires learning two generative models, namely $p(\boldsymbol{x}|\boldsymbol{c})$ and $p(\boldsymbol{x})$. However, in practice we can use the same model for this, and simply set $\boldsymbol{c} = \emptyset$ to represent the unconditional case. We then use this implicit classifier to get the following modified score function:

$$\nabla_{\boldsymbol{x}}[\log p(\boldsymbol{x}|\boldsymbol{c}) + w \log p(\boldsymbol{c}|\boldsymbol{x})] = \nabla_{\boldsymbol{x}}[\log p(\boldsymbol{x}|\boldsymbol{c}) + w(\log p(\boldsymbol{x}|\boldsymbol{c}) - \log p(\boldsymbol{x}))] \tag{25.65}$$

$$= \nabla_{\boldsymbol{x}}[(1+w) \log p(\boldsymbol{x}|\boldsymbol{c}) - w \log p(\boldsymbol{x})] \tag{25.66}$$

Larger guidance weight usually results in better individual sample quality, but lower diversity.

25.6.4 Generating high resolution images

In order to generate high resolution images, [Ho+21] proposed to use **cascaded generation**, in which we first train a model to generate 64×64 images, and then train a separate **super-resolution**

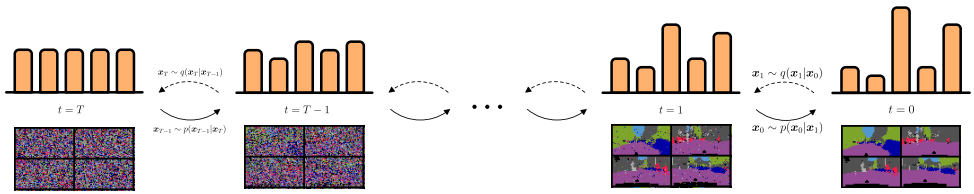


Figure 25.15: Multinomial diffusion model, applied to semantic image segmentation. The input image is on the right, and gets diffused to the noise image on the left. From Figure 1 of [Aus+21]. Used with kind permission of Emiel Hoogeboom.

model to map this to 256×256 or 1024×1024. This approach is used in Google's **Imagen** model [Sah+22b], which is a text-to-image system (see Figure 25.14). Imagen uses large pre-trained text encoder, based on T5-XXL [Raf+20a], combined with a VDM model (Section 25.2.4) based on the U-net architecture, to generate impressive-looking images (see Figure 20.2).

In addition to conditioning on text, it is possible to condition on another image to create a model for **image-to-image translation**. For example, we can learn map a gray-scale image $\boldsymbol{c}$ to a color image $\boldsymbol{x}$, or a corrupted or occluded image $\boldsymbol{c}$ to a clean version $\boldsymbol{x}$. This can be done by training a multi-task conditional diffusion model, as explained in [Sah+22a]. See Figure 20.4 for some sample outputs.

25.7 Diffusion for discrete state spaces

So far in this chapter, we have focused on Gaussian diffusion for generating real-valued data. However it is also possible to define diffusion models for discrete data, such as text or semantic segmentation labels, either by using a continuous latent embedding space (see Section 25.5.4), or by defining diffusion operations directly on the discrete state space, as we discuss beow.

25.7.1 Discrete Denoising Diffusion Probabilistic Models

In this section we discuss the Discrete Denoising Diffusion Probabilistic Model (**D3PM**) of [Aus+21], which defines a discrete time diffusion process directly on the discrete state space. (This builds on prior work such as **multinomial diffusion** [Hoo+21], and the original diffusion paper of [SD+15b].)

The basic idea is illustrated in Figure 25.15 in the context of semantic segmentation, which associates a categorical label to each pixel in an image. On the right we illustrate some sample images, and the corresponding categorical distribution that they induce over a single pixel. We gradually transform these pixel-wise distributions to the uniform distribution, using a stochastic sampling process that we describe below. We then learn a neural network to invert this process, so it can generate discrete data from noise; in the diagram, this corresponds to moving from left to right.

To ensure efficient training, we require that we can efficiently sample from $q(\boldsymbol{x}_t|\boldsymbol{x}_0)$ for an abritrary timestep t, so we can randomly sample time steps when optimizing the variational bound in Equation (25.27). In addition, we require that $q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0)$ have a tractable form, so we can efficiently compute the KL terms

$$L_{t-1}(\boldsymbol{x}_0) = \mathbb{E}_{q(\boldsymbol{x}_t|\boldsymbol{x}_0)} D_{\mathbb{KL}}\left(q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)\right) \tag{25.67}$$

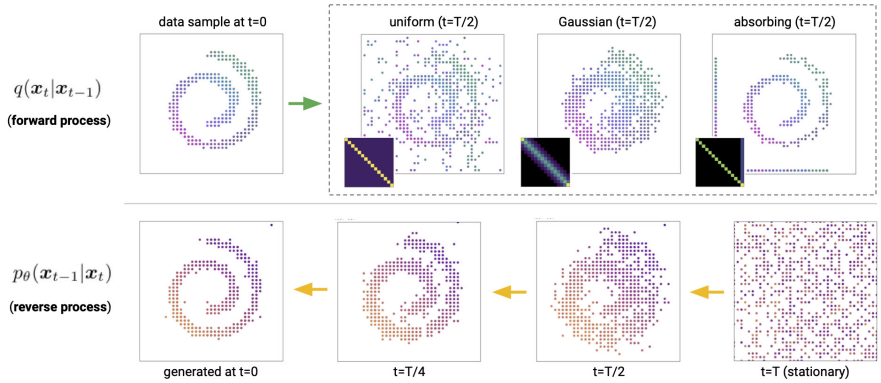


Figure 25.16: D3PM forward and (learned) reverse proccess applied to a quantized Swiss roll. Each dot represents a 2d categorical variable. Top: samples from the uniform, discretized Gaussian, and absorbing state models, along with corresponding transition matrices $\mathbf{Q}$*. Bottom: samples from a learned discretized Gaussian reverse process. From Figure 1 of [Aus+21]. Used with kind permission of Jacob Austin.*

Finally, it is useful if the forwards process converges to a known stationary distribution, $\pi(\boldsymbol{x}_T)$, which we can use for our generative prior $p(\boldsymbol{x}_T)$; this ensures $D_{\mathbb{KL}}\left(q(\boldsymbol{x}_T|\boldsymbol{x}_0) \parallel p(\boldsymbol{x}_T)\right) = 0$.

To satisfy the above criteria, we assume the state consists of D independent blocks, each representing a categorical variable, $x_t \in \{1, \ldots, K\}$; we represent this by the one-hot row vector $\boldsymbol{x}_0$. In general, this will represent a vector of probabilities. We then define the forwards diffusion kernel as follows:

$$q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}) = \text{Cat}(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}\mathbf{Q}_t) \tag{25.68}$$

where $[\mathbf{Q}_t]_{ij} = q(x_t = j|x_{t-1} = k)$ is a row stochastic transition matrix. (We discuss how to define $\mathbf{Q}_t$ in Section 25.7.2.)

We can derive the t-step marginal of the forwards process as follows:

$$q(\boldsymbol{x}_t|\boldsymbol{x}_0) = \text{Cat}(\boldsymbol{x}_t|\boldsymbol{x}_0\overline{\mathbf{Q}}_t), \ \overline{\mathbf{Q}}_t = \mathbf{Q}_1\mathbf{Q}_2\cdots\mathbf{Q}_t \tag{25.69}$$

Similarly, we can reverse the forwards process as follows:

$$q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0) = \frac{q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}, \cancel{\boldsymbol{x}_0})q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_0)}{q(\boldsymbol{x}_t|\boldsymbol{x}_0)} = \text{Cat}\left(\boldsymbol{x}_{t-1}|\frac{\boldsymbol{x}_t\mathbf{Q}_t^\top \odot \boldsymbol{x}_0\overline{\mathbf{Q}}_{t-1}}{\boldsymbol{x}_0\overline{\mathbf{Q}}_t\boldsymbol{x}_t^\top}\right) \tag{25.70}$$

We discuss how to define the generative process $p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ in Section 25.7.3. Since both distrbutions factorize, we can easily compute the KL distributions in Equation (25.67) by summing the KL for each dimension.

25.7.2 Choice of Markov transition matrices for the forward processes

In this section, we give some examples of how to represent the transition matrix $\mathbf{Q}_t$.

One simple approach is to use $\mathbf{Q}_t = (1-\beta_t)\mathbf{I} + \beta_t/K$, which we can write in scalar form as follows:

$$[\mathbf{Q}_t]_{ij} = \begin{cases} 1 - \frac{K-1}{K}\beta_t & \text{if } i = j \\ \frac{1}{K}\beta_t & \text{if } i \neq j \end{cases} \tag{25.71}$$

Intuïtively, this adds a little amount of uniform noise over the K classes, and with a large probability, $1-\beta_t$, we sample from $\boldsymbol{x}_{t-1}$. We call this the uniform kernel. Since this is a doubly stochastic matrix with strictly positive entries, the stationary distributon is uniform. See Figure 25.16 for an illustration.

In the case of the uniform kernel, one can show [Hoo+21] that the marginal distribution is given by

$$q(\boldsymbol{x}_t|\boldsymbol{x}_0) = \text{Cat}(\boldsymbol{x}_t|\overline{\alpha}_t\boldsymbol{x}_0 + (1-\overline{\alpha}_t)/K) \tag{25.72}$$

where $\alpha_t = 1-\beta_t$ and $\overline{\alpha}_t = \prod_{\tau=1}^{t}\alpha_\tau$. This is similar to the Gaussian case discussed in Section 25.2. Furthermore, we can derive the posterior distribution as follows:

$$q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0) = \text{Cat}(\boldsymbol{x}_{t-1}|\boldsymbol{\theta}_{\text{post}}(\boldsymbol{x}_t, \boldsymbol{\theta}_0)),\ \boldsymbol{\theta}_{\text{post}}(\boldsymbol{x}_t, \boldsymbol{\theta}_0) = \tilde{\boldsymbol{\theta}}/\sum_{k=1}^{K}\tilde{\theta}_k \tag{25.73}$$

$$\tilde{\boldsymbol{\theta}} = [\alpha_t\boldsymbol{x}_t + (1-\alpha_t)/K] \odot [\overline{\alpha}_{t-1}\boldsymbol{x}_0 + (1-\overline{\alpha}_{t-1})/K] \tag{25.74}$$

Another option is to define a special **absorbing state** m, representing a MASK token, which we transition into with probability β_t. Formally, we have $\mathbf{Q}_t = (1-\beta_t)\mathbf{I} + \beta_t\mathbf{1}\boldsymbol{e}_m^\mathsf{T}$, or, in scalar form,

$$[\mathbf{Q}_t]_{ij} = \begin{cases} 1 & \text{if } i = j = m \\ 1-\beta_t & \text{if } i = j \neq m \\ \beta_t & \text{if } j = m, i \neq m \end{cases} \tag{25.75}$$

This converges to a point-mass distribution on state m. See Figure 25.16 for an illustration.

Another option, suitable for quantized ordinal values, is to use a **discretized Gaussian**, that transitions to other nearby states, with a probability that depends on how similar the states are in numerical value. If we ensure the transition matrix is doubly stochastic, the resulting stationary distribution will again be uniform. See Figure 25.16 for an illustration.

25.7.3 Parameterization of the reverse process

While it is possible to directly predict the logits $p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ using a neural network $f_{\boldsymbol{\theta}}(\boldsymbol{x}_t)$, it is preferable to directly predict the logits of the output, using $\tilde{p}_{\boldsymbol{\theta}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t)$; we can then combine this with the analytical expression for $q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0)$ to get

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) \propto \sum_{\tilde{\boldsymbol{x}}_0} q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \tilde{\boldsymbol{x}}_0)\tilde{p}_{\boldsymbol{\theta}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t) \tag{25.76}$$

(The sum over $\tilde{\boldsymbol{x}}_0$ takes $O(DK)$ time, if there are D dimensions, each with K values.) One advantage of this approach, compared to directly learning $p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$, is that the model will automatically

satisfy any sparsity constraints in $\mathbf{Q}_t$. In addition, we can perform inference with k steps at a time, by predicting

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-k}|\boldsymbol{x}_t) \propto \sum_{\tilde{\boldsymbol{x}}_0} q(\boldsymbol{x}_{t-k}|\boldsymbol{x}_t, \tilde{\boldsymbol{x}}_0)\tilde{p}_{\boldsymbol{\theta}}(\tilde{\boldsymbol{x}}_0|\boldsymbol{x}_t) \tag{25.77}$$

Note that, in the multi-step Gaussian case, we require more complex models to handle multimodaility (see Section 25.5.2). By contrast, discrete distributions already have this flexibility built in.

25.7.4 Noise schedules

In this section we discuss how to choose the noise schedule for β_t. For discretized Gaussian diffusion, [Aus+21] propose to linearly increase the variance of the Gaussian noise before the discretization step. For uniform diffusion, we can use a cosine schedule of the form $\alpha_t = \cos(\frac{t/T+s}{1+s}\frac{\pi}{2})$, where $s = 0.08$, as proposed in [ND21]. (Recall that $\beta_t = 1 - \alpha_t$, so the noise increases over time.) For masked diffusion, we can use a schedule of the form $\beta_t = 1/(T - t + 1)$, as proposed in [SD+15b].

25.7.5 Connections to other probabilistic models for discrete sequences

There are interesting connections between D3PM and other probabilistic text models. For example, suppose we define the transition matrix as a combination of the unifrom transition matrix and an absorbing MASK state, i.e., $\mathbf{Q} = \alpha\mathbf{1}\boldsymbol{e}_m^\mathsf{T} + \beta\mathbf{1}\mathbf{1}^\mathsf{T}/K + (1-\alpha-\beta)\mathbf{I}$. For a one-step diffusion process in which $q(\boldsymbol{x}_1|\boldsymbol{x}_0)$ replaces $\alpha = 10\%$ of the tokens with MASK, and $\beta = 5\%$ uniformly at random, we recover the same objective that is used to train the **BERT** language model, namely

$$L_0(\boldsymbol{x}_0) = -\mathbb{E}_{q(\boldsymbol{x}_1|\boldsymbol{x}_0)} \log p_{\boldsymbol{\theta}}(\boldsymbol{x}_0|\boldsymbol{x}_1) \tag{25.78}$$

(This follows since $L_T = 0$, and there are no other time steps used in the variational bound in Equation (25.27).)

Now consider a diffusion process that deterministically masks tokens one-by-one. For a sequence of length $N = T$, we have $q([\boldsymbol{x}_t]_i|\boldsymbol{x}_0) = [\boldsymbol{x}_0]_i$ if $i < N - t$ (pass through), else $[\boldsymbol{x}_t]_i$ is set to MASK. Because this is a deterministic process, the posterior $q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0)$ is a delta function on the $\boldsymbol{x}_t$ with one fewer mask tokens. One can then show that the KL term becomes $D_{\mathbb{KL}}\left(q([\boldsymbol{x}_t]_i|\boldsymbol{x}_t, \boldsymbol{x}_0) \parallel p_{\boldsymbol{\theta}}([\boldsymbol{x}_{t-1}]_i|\boldsymbol{x}_t)\right) = -\log p_{\boldsymbol{\theta}}([\boldsymbol{x}_0]_i|\boldsymbol{x}_t)$, which is the standard cross-entropy loss for an autoregressive model.

Finally one can show that generative **masked language models**, such as [WC19; Gha+19], also correspond to discrete diffusion processes: the sequence starts will all locations masked out, and each step, a set of tokens are generated, given the previous sequence. The **MaskGIT** method of [Cha+22] uses a similar procedure in the image domain, after applying vector quantization to image patches. These parallel, iterative decoders are much faster than sequential autoregressive decoders. See Figure 25.17 for an illustration.

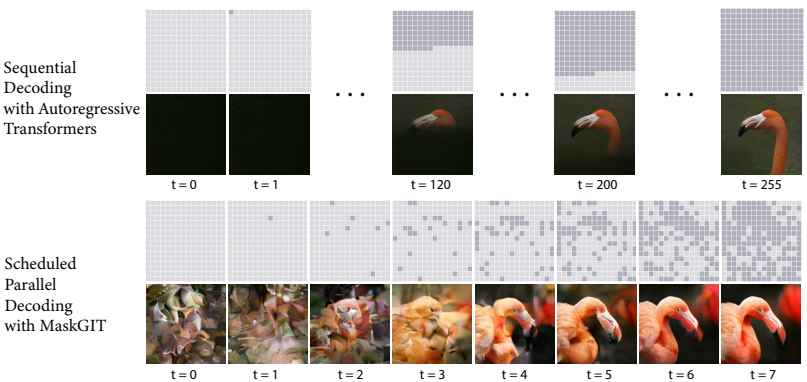


Figure 25.17: Comparison of sequential image generation with a transformer (top) vs parallel generation with MaskGIT (bottom). All pixels start out in the MASK state, denoted by light gray. In the transformer, we generate one pixel at a time, so it takes 256 steps for the whole image. In the MaskGIT method, multiple states are generated in parallel, which only takes 8 steps. From Figure 2 of [Cha+22]. Used with kind permission of Huiwen Chang.

26 Generative adversarial networks

This chapter is written by Mihaela Rosca, Shakir Mohamed, and Balaji Lakshminarayanan.

26.1 Introduction

In this chapter, we focus on **implicit generative models**, which are a kind of probabilistic model without an explicit likelihood function [ML16]. This includes the family of **generative adversarial networks** or **GANs** [Goo16]. In this chapter, we provide an introduction to this topic, focusing on a probabilistic perspective.

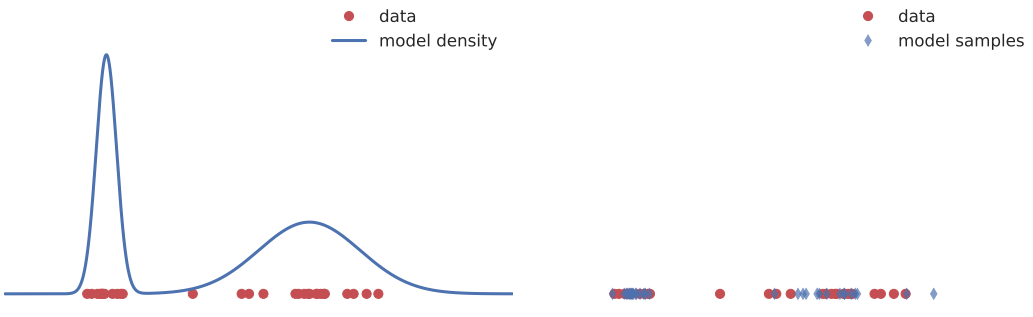
To develop a probabilistic formulation for GANs, it is useful to first distinguish between two types of probabilistic models: "**prescribed probabilistic models**" and "**implicit probabilistic models**" [DG84]. Prescribed probabilistic models, which we will call **explicit probabilistic models**, provide an explicit parametric specification of the distribution of an observed random variable $\boldsymbol{x}$, specifying a log-likelihood function $\log q_\theta(\boldsymbol{x})$ with parameters $\boldsymbol{\theta}$. Most models we encountered in this book thus far are of this form, whether they be state-of-the-art classifiers, large-vocabulary sequence models, or fine-grained spatio-temporal models. Alternatively, we can specify an **implicit probabilistic model** that defines a stochastic procedure to directly generate data. Such models are the natural approach for problems in climate and weather, population genetics, and ecology, since the mechanistic understanding of such systems can be used to directly describe the generative model. We illustrate the difference between implicit and explicit models in Figure 26.1.

The form of implicit generative models we focus on in this chapter can be expressed as a probabilistic latent variable model, similar to VAEs (Chapter 21). Implicit generative models use a latent variable $\boldsymbol{z}$ and transform it using a deterministic function $G_{\boldsymbol{\theta}}$ that maps from $\mathbb{R}^m \to \mathbb{R}^d$ using parameters $\boldsymbol{\theta}$. Implicit generative models do not include a likelihood function or observation model. Instead, the generating procedure defines a valid density on the output space that forms an effective likelihood function:

$$\boldsymbol{x} = G_{\boldsymbol{\theta}}(\boldsymbol{z}'); \qquad \boldsymbol{z}' \sim q(\boldsymbol{z}) \tag{26.1}$$

$$q_\theta(\boldsymbol{x}) = \frac{\partial}{\partial x_1} \cdots \frac{\partial}{\partial x_d} \int_{\{G_{\boldsymbol{\theta}}(\boldsymbol{z}) \leq \boldsymbol{x}\}} q(\boldsymbol{z}) d\boldsymbol{z}, \tag{26.2}$$

where $q(\boldsymbol{z})$ is a distribution over latent variables that provides the external source of randomness. Equation (26.2) is the definition of the transformed density $q_\theta(\boldsymbol{x})$ defined as the derivative of a cumulative distribution function, and hence integrates the distribution $q(\boldsymbol{z})$ over all events defined



(a) Prescribed generative model. *(b) Implicit generative model.*

Figure 26.1: Visualizing the difference between prescribed and implicit generative models. Prescribed models provide direct access to the learned density (sometimes unnormalized). Implicit models only provide access to a simulator which can be used to generate samples from an implied density. Generated by genmo_types_implicit_explicit.ipynb

by the set $\{G_{\boldsymbol{\theta}}(\boldsymbol{z}) \leq \boldsymbol{x}\}$. When the latent and data dimension are equal ($m = d$) and the function $G_{\boldsymbol{\theta}}(\boldsymbol{z})$ is invertible or has easily characterized roots, we recover the rule for transformations of probability distributions. This transformation of variables property is also used in normalizing flows (Chapter 23). In diffusion models (Chapter 25), we also transform noise into data and vice versa, but the transformation is not strictly invertible.

We can develop more general and flexible implicit generative models where the function G is a non-linear function with $d > m$, e.g., specified by a deep network. Such models are sometimes called **generator networks** or **generative neural samplers**; they can also be throught of as **differentiable simulators**. Unfortunately the integral (26.2) is intractable in these kinds of models, and we may not even be able to determine the set $\{G_{\boldsymbol{\theta}}(\boldsymbol{z}) \leq \boldsymbol{x}\}$. Of course, intractability is also a challenge for explicit latent variable models such as VAEs (Chapter 21), but in the GAN case, the lack of a likelihood term makes the learning problem even harder. Therefore this problem is called **likelihood-free inference** or **simulation-based inference**.

Likelihood-free inference also forms the basis of the field known as **approximate Bayesian computation** or **ABC**, which we briefly discuss in Section 13.6.5. ABC and GANs give us two different algorithmic frameworks for learning in implicit generative models. Both approaches rely on a learning principle based on *comparing real and simulated data*. This type of learning by comparison instantiates a core principle of likelihood-free inference, and expanding on this idea is the focus of the next section. The subsequent sections will then focus on GANs specifically, to develop a more detailed foundation and practical considerations. (See also https://poloclub.github.io/ganlab/ for an interactive tutorial.)

26.2 Learning by comparison

In most of this book, we rely on the principle of maximum likelihood for learning. By maximizing the likelihood we effectively minimize the KL divergence between the model q_θ (with parameters $\boldsymbol{\theta}$)

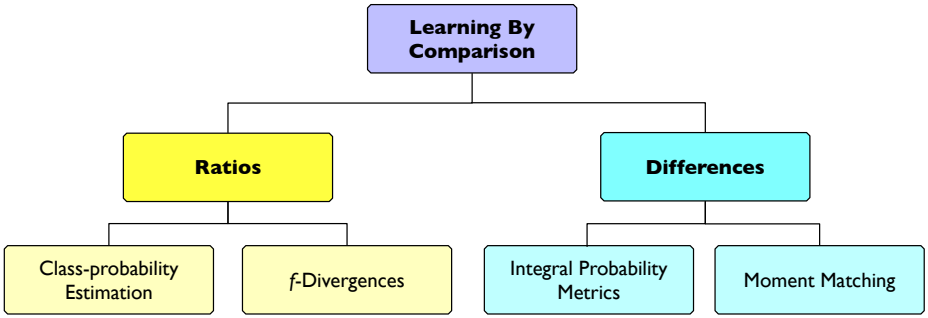


Figure 26.2: Overview of approaches for learning in implicit generative models

and the unknown true data distribution p^*. Recalling equation (26.2), in implicit models we cannot evaluate $q_\theta(\boldsymbol{x})$, and thus cannot use maximum likelihood training. As implicit models provide a sampling procedure, we instead are searching for learning principles that only use *samples from the model*.

The task of learning in implicit models is to determine, from two sets of samples, whether their distributions are close to each other and to quantify the distance between them. We can think of this as a 'two sample' or likelihood-free approach to learning by comparison. There are many ways of doing this, including using distributional divergences or distances through binary classification, the method of moments, and other approaches. Figure 26.2 shows an overview of different approaches for learning by comparison.

26.2.1 Guiding principles

We are looking for objectives $\mathcal{D}(p^*, q)$ that satisfy the following requirements:

1. Provide guarantees about learning the data distribution: $\text{argmin}_q \mathcal{D}(p^*, q) = p^*$.
2. Can be evaluated only using samples from the data and model distribution.
3. Are computationally cheap to evaluate.

Many distributional distances and divergences satisfy the first requirement, since by definition they satisfy the following:

$$\mathcal{D}(p^*, q) \geq 0; \qquad \mathcal{D}(p^*, q) = 0 \iff p^* = q \tag{26.3}$$

Many distributional distances and divergences, however, fail to satisfy the other two requirements: they cannot be evaluated only using samples — such as the KL divergence, or are computationally intractable — such as the Wasserstein distance. The main approach to overcome these challenges is to *approximate the desired quantity through optimization* by introducing a comparison model, often called a **discriminator** or a **critic** D, such that:

$$\mathcal{D}(p^*, q) = \underset{D}{\text{argmax}}\, \mathcal{F}(D, p^*, q) \tag{26.4}$$

where $\mathcal{F}$ is a functional that depends on p^* and q only through samples. For the cases we discuss, both the model and the critic are parametric with parameters $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ respectively; instead of optimizing over distributions or functions, we optimize with respect to parameters. For the critic, this results in the optimization problem $\text{argmax}_{\boldsymbol{\phi}} \mathcal{F}(D_{\boldsymbol{\phi}}, p^*, q_\theta)$. For the model parameters $\boldsymbol{\theta}$, the exact objective $\mathcal{D}(p^*, q_\theta)$ is replaced with the tractable approximation provided through the use of $D_{\boldsymbol{\phi}}$.

A convenient approach to ensure that $\mathcal{F}(D_{\boldsymbol{\phi}}, p^*, q_\theta)$ can be estimated using only samples from the model and the unknown data distribution is to depend on the two distributions only in expectation:

$$\mathcal{F}(D_{\boldsymbol{\phi}}, p^*, q_\theta) = \mathbb{E}_{p^*(\boldsymbol{x})} f(\boldsymbol{x}, \boldsymbol{\phi}) + \mathbb{E}_{q_\theta(\boldsymbol{x})} g(\boldsymbol{x}, \boldsymbol{\phi}) \tag{26.5}$$

where f and g are real valued functions whose choice will define $\mathcal{F}$. In the case of implicit generative models, this can be rewritten to use the sampling path $\boldsymbol{x} = G_{\boldsymbol{\theta}}(\boldsymbol{z}), \quad \boldsymbol{z} \sim q(\boldsymbol{z})$:

$$\mathcal{F}(D_{\boldsymbol{\phi}}, p^*, q_\theta) = \mathbb{E}_{p^*(\boldsymbol{x})} f(\boldsymbol{x}, \boldsymbol{\phi}) + \mathbb{E}_{q(\boldsymbol{z})} g(G_{\boldsymbol{\theta}}(\boldsymbol{z}), \boldsymbol{\phi}) \tag{26.6}$$

which can be estimated using Monte Carlo estimation

$$\mathcal{F}(D_{\boldsymbol{\phi}}, p^*, q_\theta) \approx \frac{1}{N} \sum_{i=1}^{N} f(\widehat{\boldsymbol{x}}_i, \boldsymbol{\phi}) + \frac{1}{M} \sum_{i=1}^{M} g(G_{\boldsymbol{\theta}}(\widehat{\boldsymbol{z}}_i), \boldsymbol{\phi}); \qquad \widehat{\boldsymbol{x}}_i \sim p^*(\boldsymbol{x}); \qquad \widehat{\boldsymbol{z}}_i \sim q(\boldsymbol{z}) \tag{26.7}$$

Next, we will see how to instantiate these guiding principles in order to find the functions f and g and thus the objective $\mathcal{F}$ which can be used to train implicit models: class probability estimation (Section 26.2.2), bounds on f-divergences (Section 26.2.3), integral probability metrics (Section 26.2.4), and moment matching (Section 26.2.5).

26.2.2 Density ratio estimation using binary classifiers

One way to compare two distributions p^* and q_θ is to compute their density ratio $r(\boldsymbol{x}) = \frac{p^*(\boldsymbol{x})}{q_\theta(\boldsymbol{x})}$. The distributions are the same if and only if the ratio is 1 everywhere in the support of $q_{\boldsymbol{\theta}}$. Since we cannot evaluate the densities of implicit models, we must instead develop techniques to compute the density ratio from samples alone, following the guiding principles established above.

Fortunately, we can use the trick from Section 2.7.5 which converts density estimation into a binary classification problem to write

$$\frac{p^*(\boldsymbol{x})}{q_\theta(\boldsymbol{x})} = \frac{D(\boldsymbol{x})}{1 - D(\boldsymbol{x})} \tag{26.8}$$

where $D(\boldsymbol{x})$ is the discriminator or critic which is trained to distinguish samples coming from p^* vs $q_{\boldsymbol{\theta}}$.

For parametric classification, we can learn discriminators $D_{\boldsymbol{\phi}}(\boldsymbol{x}) \in [0, 1]$ with parameters $\boldsymbol{\phi}$. Using knowledge and insight about probabilistic classification, we can learn the parameters by minimizing any proper scoring rule [GR07b] (see also Section 14.2.1). For the familiar Bernoulli log-loss (or binary cross entropy loss), we obtain the objective:

$$\begin{aligned} V(q_\theta, p^*) &= \arg\max_{\boldsymbol{\phi}} \mathbb{E}_{p(\boldsymbol{x}|y)p(y)}[y \log D_{\boldsymbol{\phi}}(\boldsymbol{x}) + (1-y)\log(1 - D_{\boldsymbol{\phi}}(\boldsymbol{x}))] \\ &= \arg\max_{\boldsymbol{\phi}} \mathbb{E}_{p(\boldsymbol{x}|y=1)p(y=1)} \log D_{\boldsymbol{\phi}}(\boldsymbol{x}) + \mathbb{E}_{p(\boldsymbol{x}|y=0)p(y=0)} \log(1 - D_{\boldsymbol{\phi}}(\boldsymbol{x})) \qquad (26.9) \\ &= \arg\max_{\boldsymbol{\phi}} \frac{1}{2}\mathbb{E}_{p^*(\boldsymbol{x})} \log D_{\boldsymbol{\phi}}(\boldsymbol{x}) + \frac{1}{2}\mathbb{E}_{q_\theta(\boldsymbol{x})} \log(1 - D_{\boldsymbol{\phi}}(\boldsymbol{x})). \qquad (26.10) \end{aligned}$$

Loss	**Objective Function** ($D := D(\boldsymbol{x};\boldsymbol{\phi}) \in [0,1]$)
Bernoulli loss	$\mathbb{E}_{p^*(\boldsymbol{x})}[\log D] + \mathbb{E}_{q_\theta(\boldsymbol{x})}[\log(1-D)]$
Brier score	$\mathbb{E}_{p^*(\boldsymbol{x})}[-(1-D)^2] + \mathbb{E}_{q_\theta(\boldsymbol{x})}[-D^2]$
Exponential loss	$\mathbb{E}_{p^*(\boldsymbol{x})}\left[\left(-\frac{1-D}{D}\right)^{\frac{1}{2}}\right] + \mathbb{E}_{q_\theta(\boldsymbol{x})}\left[\left(-\frac{D}{1-D}\right)^{\frac{1}{2}}\right]$
Misclassification	$\mathbb{E}_{p^*(\boldsymbol{x})}[-\mathbb{I}[D \leq 0.5]] + \mathbb{E}_{q_\theta(\boldsymbol{x})}[-\mathbb{I}[D > 0.5]]$
Hinge loss	$\mathbb{E}_{p^*(\boldsymbol{x})}\left[-\max\left(0, 1-\log\frac{D}{1-D}\right)\right] + \mathbb{E}_{q_\theta(\boldsymbol{x})}\left[-\max\left(0, 1+\log\frac{D}{1-D}\right)\right]$
Spherical	$\mathbb{E}_{p^*(\boldsymbol{x})}[\alpha D] + \mathbb{E}_{q_\theta(\boldsymbol{x})}[\alpha(1-D)]; \quad \alpha = (1-2D+2D^2)^{-\frac{1}{2}}$

Table 26.1: Proper scoring rules that can be maximized in class probability-based learning of implicit generative models. Based on [ML16].

The same procedure can be extended beyond the Bernoulli log-loss to other proper scoring rules used for binary classification, such as those presented in Table 26.1, adapted from [ML16]. The optimal discriminator D is $\frac{p^*(\boldsymbol{x})}{p^*(\boldsymbol{x})+q_\theta(\boldsymbol{x})}$, since:

$$\frac{p^*(\boldsymbol{x})}{q_\theta(\boldsymbol{x})} = \frac{D^*(\boldsymbol{x})}{1-D^*(\boldsymbol{x})} \implies D^*(\boldsymbol{x}) = \frac{p^*(\boldsymbol{x})}{p^*(\boldsymbol{x})+q_\theta(\boldsymbol{x})} \tag{26.11}$$

By substituting the optimal discriminator into the scoring rule (26.10), we can show that the objective V can also be interpreted as the minimization of the Jensen-Shannon divergence.

$$V^*(q_\theta, p^*) = \frac{1}{2}\mathbb{E}_{p^*(\boldsymbol{x})}[\log \frac{p^*(\boldsymbol{x})}{p^*(\boldsymbol{x})+q_\theta(\boldsymbol{x})}] + \frac{1}{2}\mathbb{E}_{q_\theta(\boldsymbol{x})}[\log(1 - \frac{p^*(\boldsymbol{x})}{p^*(\boldsymbol{x})+q_\theta(\boldsymbol{x})})] \tag{26.12}$$

$$= \frac{1}{2}\mathbb{E}_{p^*(\boldsymbol{x})}[\log \frac{p^*(\boldsymbol{x})}{\frac{p^*(\boldsymbol{x})+q_\theta(\boldsymbol{x})}{2}}] + \frac{1}{2}\mathbb{E}_{q_\theta(\boldsymbol{x})}[\log(\frac{q_\theta(\boldsymbol{x})}{\frac{p^*(\boldsymbol{x})+q_\theta(\boldsymbol{x})}{2}})] - \log 2 \tag{26.13}$$

$$= \frac{1}{2}D_{\mathbb{KL}}\left(p^* \,\|\, \frac{p^*+q_\theta}{2}\right) + \frac{1}{2}D_{\mathbb{KL}}\left(q_\theta \,\|\, \frac{p^*+q_\theta}{2}\right) - \log 2 \tag{26.14}$$

$$= JSD(p^*, q_\theta) - \log 2 \tag{26.15}$$

where JSD denotes the Jensen-Shannon divergence:

$$JSD(p^*, q_\theta) = \frac{1}{2}D_{\mathbb{KL}}\left(p^* \,\|\, \frac{p^*+q_\theta}{2}\right) + \frac{1}{2}D_{\mathbb{KL}}\left(q_\theta \,\|\, \frac{p^*+q_\theta}{2}\right) \tag{26.16}$$

This establishes a connection between *optimal* binary classification and distributional divergences. By using binary classification, we were able to compute the distributional divergence using only samples, which is the important property needed for learning implicit generative models; as expressed in the guiding principles (Section 26.2.1), we have turned an intractable estimation problem — how to estimate the JSD divergence, into an optimization problem — how to learn a classifier which can be used to approximate that divergence.

We would like to train the parameters $\boldsymbol{\theta}$ of generative model to minimize the divergence:

$$\min_{\boldsymbol{\theta}} JSD(p^*, q_\theta) = \min_{\boldsymbol{\theta}} V^*(q_\theta, p^*) + \log 2 \tag{26.17}$$

$$= \min_{\boldsymbol{\theta}} \frac{1}{2}\mathbb{E}_{p^*(\boldsymbol{x})} \log D^*(\boldsymbol{x}) + \frac{1}{2}\mathbb{E}_{q_\theta(\boldsymbol{x})} \log(1 - D^*(\boldsymbol{x})) + \log 2 \tag{26.18}$$

Since we do not have access to the optimal classifier D^* but only to the neural approximation $D_{\boldsymbol{\phi}}$ obtained using the optimization in (26.10) , this results in a min-max optimization problem:

$$\min_{\boldsymbol{\theta}} \max_{\boldsymbol{\phi}} \frac{1}{2}\mathbb{E}_{p^*(\boldsymbol{x})}[\log D_{\boldsymbol{\phi}}(\boldsymbol{x})] + \frac{1}{2}\mathbb{E}_{q_\theta(\boldsymbol{x})}[\log(1 - D_{\boldsymbol{\phi}}(\boldsymbol{x}))] \tag{26.19}$$

By replacing the generating procedure (26.1) in (26.19) we obtain the objective in terms of the latent variables $\boldsymbol{z}$ of the implicit generative model:

$$\min_{\boldsymbol{\theta}} \max_{\boldsymbol{\phi}} \frac{1}{2}\mathbb{E}_{p^*(\boldsymbol{x})}[\log D_{\boldsymbol{\phi}}(\boldsymbol{x})] + \frac{1}{2}\mathbb{E}_{q(\boldsymbol{z})}[\log(1 - D_{\boldsymbol{\phi}}(G_{\boldsymbol{\theta}}(\boldsymbol{z})))], \tag{26.20}$$

which recovers the definition proposed in the original GAN paper [Goo+14]. The core principle behind GANs is to train a discriminator, in this case a binary classifier, to approximate a distance or divergence between the model and data distributions, and to then train the generative model to minimize this approximation of the divergence or distance.

Beyond the use of the Bernoulli scoring rule used above, other scoring rules have been used to train generative models via min-max optimization. The Brier scoring rule, which under discriminator optimality conditions can be shown to correspond to minimizing the Pearson χ^2 divergence via similar arguments as the ones shown above has lead to LS-GAN [Mao+17]. The hinge scoring rule has become popular [Miy+18b; BDS18], and under discriminator optimality conditions corresponds to minimizing the total variational distance [NWJ+09].

The connection between proper scoring rules and distributional divergences allows the construction of convergence guarantees for the learning criteria above, under infinite capacity of the discriminator and generator: since the minimizer of distributional divergence is the true data distribution (Equation 26.3), if the discriminator is optimal and the generator has enough capacity, it will learn the data distribution. In practice however, this assumption will not hold, as discriminators are rarely optimal; we will discuss this at length in Section 26.3.

26.2.3 Bounds on f-divergences

As we saw with the appearance of the Jensen-Shannon divergence in the previous section, we can consider directly using a measure of distributional divergence to derive methods for learning in implicit models. One general class of divergences are the f-divergences (Section 2.7.1) defined as:

$$\mathcal{D}_f\left[p^*(\boldsymbol{x}) \| q_\theta(\boldsymbol{x})\right] = \int q_\theta(\boldsymbol{x}) f\left(\frac{p^*(\boldsymbol{x})}{q_\theta(\boldsymbol{x})}\right) d\boldsymbol{x} \tag{26.21}$$

where f is a convex function such that $f(1) = 0$. For different choices of f, we can recover known distributional divergences such as the KL, reverse KL, and Jensen-Shannon divergence. We discuss such connections in Section 2.7.1, and provide a summary in Table 26.2.

To evaluate Equation (26.21) we will need to evaluate the density of the data $p^*(\boldsymbol{x})$ and the model $q_\theta(\boldsymbol{x})$, neither of which are available. In the previous section we overcame the challenge of evaluating the density ratio by transforming it into a problem of binary classification. In this section, we will instead look towards the role of lower bounds on f-divergences, which is an approach for tractability that is also used for variational inference (Chapter 10).

Divergence	f	$f^\dagger$	Optimal Critic
KL	$u \log u$	e^{u-1}	$1 + \log r(\boldsymbol{x})$
Reverse KL	$-\log u$	$-1 - \log(-u)$	$-1/r(\boldsymbol{x})$
JSD	$u \log u - (u+1) \log \frac{u+1}{2}$	$-\log(2 - e^u)$	$\frac{2}{1+1/r(\boldsymbol{x})})$
Pearson χ^2	$(u-1)^2$	$\frac{1}{4}u^2 + u$	$\left(\sqrt{r(\boldsymbol{x})} - 1\right)\sqrt{1/r(\boldsymbol{x})}$

Table 26.2: Standard divergences as f divergences for various choices of f. The optimal critic is written as a function of the density ratio $r(\boldsymbol{x}) = \frac{p^*(\boldsymbol{x})}{q_\theta(\boldsymbol{x})}$.

f-divergences have a widely-developed theory in convex analysis and information theory. Since the function f in Equation (26.21) is convex, we know that we can find a tangent that bounds it from below. The variational formulation of the f-divergence is [NWJ10b; NCT16c]:

$$\mathcal{D}_f\left[p^*(\boldsymbol{x}) \| q_\theta(\boldsymbol{x})\right] = \int q_\theta(\boldsymbol{x}) f\left(\frac{p^*(\boldsymbol{x})}{q_\theta(\boldsymbol{x})}\right) d\boldsymbol{x} \tag{26.22}$$

$$= \int q_\theta(\boldsymbol{x}) \sup_{t:\mathcal{X}\to\mathbb{R}} \left[t(\boldsymbol{x})\frac{p^*(\boldsymbol{x})}{q_\theta(\boldsymbol{x})} - f^\dagger(t(\boldsymbol{x}))\right] d\boldsymbol{x} \tag{26.23}$$

$$= \int \sup_{t:\mathcal{X}\to\mathbb{R}} p^*(\boldsymbol{x}) t(\boldsymbol{x}) - q_\theta(\boldsymbol{x}) f^\dagger(t(\boldsymbol{x})) d\boldsymbol{x} \tag{26.24}$$

$$\geq \sup_{t\in\mathcal{T}} \mathbb{E}_{p^*(\boldsymbol{x})}[t(\boldsymbol{x})] - \mathbb{E}_{q_\theta(\boldsymbol{x})}[f^\dagger(t(\boldsymbol{x}))]. \tag{26.25}$$

In the second line we use the result from convex analysis, discussed Supplementary Section 6.3, that re-expresses the convex function f using $f(u) = \sup_t ut - f^\dagger(t)$, where $f^\dagger$ is the convex conjugate of the function f, and t is a parameter we optimize over. Since we apply f at $u = \frac{p^*(\boldsymbol{x})}{q_\theta(\boldsymbol{x})}$ for all $\boldsymbol{x} \in \mathcal{X}$, we make the parameter t be a function $t(\boldsymbol{x})$. The final inequality comes from replacing the supremum over all functions from the data domain $\mathcal{X}$ to $\mathbb{R}$ with the supremum over a family of functions $\mathcal{T}$ (such as the family of functions expressible by a neural network architecture), which might not be able to capture the true supremum. The function t takes the role of the discriminator or critic.

The final expression in Equation (26.25) follows the general desired form of Equation 26.5: it is the difference of two expectations, and these expectations can be computed by Monte Carlo estimation using only samples, as in Equation (26.7); despite starting with an objective (Equation 26.21) which contravened the desired principles for training implicit generative models, variational bounds have allowed us to construct an approximation which satisfies all desiderata.

Using bounds on the f-divergence, we obtain an objective (26.25) that allows learning both the generator and critic parameters. We use a critic D with parameters $\boldsymbol{\phi}$ to estimate the bound, and then optimize the parameters $\boldsymbol{\theta}$ of the generator to minimize the approximation of the f-divergence provided by the critic (we replace t above with $D_{\boldsymbol{\phi}}$, to retain standard GAN notation):

$$\min_{\boldsymbol{\theta}} \mathcal{D}_f(p^*, q_\theta) \geq \min_{\boldsymbol{\theta}} \max_{\boldsymbol{\phi}} \mathbb{E}_{p^*(\boldsymbol{x})}[D_{\boldsymbol{\phi}}(\boldsymbol{x})] - \mathbb{E}_{q_\theta(\boldsymbol{x})}[f^\dagger(D_{\boldsymbol{\phi}}(\boldsymbol{x}))] \tag{26.26}$$

$$= \min_{\boldsymbol{\theta}} \max_{\boldsymbol{\phi}} \mathbb{E}_{p^*(\boldsymbol{x})}[D_{\boldsymbol{\phi}}(\boldsymbol{x})] - \mathbb{E}_{q(\boldsymbol{z})}[f^\dagger(D_{\boldsymbol{\phi}}(G_{\boldsymbol{\theta}}(\boldsymbol{z})))] \tag{26.27}$$

This approach to train an implicit generative model leads to f-GANs [NCT16c]. It is worth noting that there exists an equivalence between the scoring rules in the previous section and bounds on

f-divergences [RW11]: for each scoring rule we can find an f-divergence that leads to the same training criteria and the same min-max game of Equation 26.27. An intuitive way to grasp the connection between f-divergences and proper scoring rules is through their use of density ratios: in both cases the optimal critic approximates a quantity directly related to the density ratio (see Table 26.2 for f-divergences and Equation (26.11) for scoring rules).

26.2.4 Integral probability metrics

Instead of comparing distributions by using their ratio as we did in the previous two sections, we can instead study their difference. A general class of measure of difference is given by the Integral Probability Metrics (Section 2.7.2) defined as:

$$I_{\mathcal{F}}(p^*(\boldsymbol{x}), q_\theta(\boldsymbol{x})) = \sup_{f \in \mathcal{F}} \left| \mathbb{E}_{p^*(\boldsymbol{x})} f(\boldsymbol{x}) - \mathbb{E}_{q_\theta(\boldsymbol{x})} f(\boldsymbol{x}) \right|. \tag{26.28}$$

The function f is a test or witness function that will take the role of the discriminator or critic. To use IPMs we must define the class of real valued, measurable functions $\mathcal{F}$ over which the supremum is taken, and this choice will lead to different distances, just as choosing different convex functions f leads to different f-divergences. Integral probability metrics are distributional distances: beyond satisfying the conditions for distributional divergences $\mathcal{D}(p^*, q) \geq 0; \quad \mathcal{D}(p^*, q) = 0 \iff p^* = q$ (Equation (26.3)), they are also symmetric $\mathcal{D}(p, q) = \mathcal{D}(q, p)$ and satisfy the triangle inequality $\mathcal{D}(p, q) \leq \mathcal{D}(p, r) + \mathcal{D}(r, q)$.

Not all function families satisfy these conditions of create a valid distance $I_{\mathcal{F}}$. To see why consider the case where $\mathcal{F} = \{z\}$ where z is the function $z(\boldsymbol{x}) = 0$. This choice of $\mathcal{F}$ entails that regardless of the two distributions chosen, the value in Equation 26.28 would be 0, violating the requirement that distance between two distributions be 0 only if the two distributions are the same. A popular choice of $\mathcal{F}$ for which $I_{\mathcal{F}}$ satisfies the conditions of a valid distributional distance is the set of 1-Lipschitz functions, which leads to the Wasserstein distance [Vil08]:

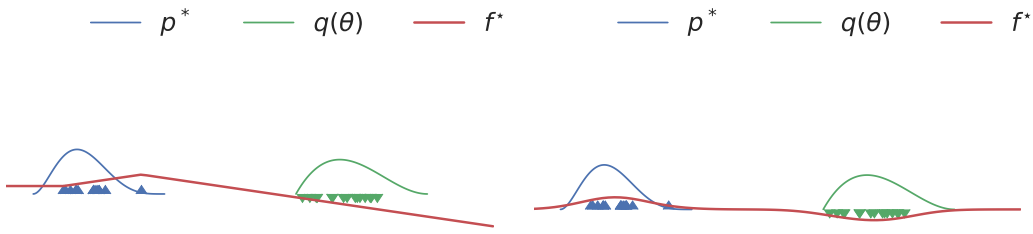
$$W_1(p^*(\boldsymbol{x}), q_\theta(\boldsymbol{x})) = \sup_{f : \|f\|_{\text{Lip}} \leq 1} \mathbb{E}_{p^*(\boldsymbol{x})} f(\boldsymbol{x}) - \mathbb{E}_{q_\theta(\boldsymbol{x})} f(\boldsymbol{x}) \tag{26.29}$$

We show an example of a Wasserstein critic in Figure 26.3a. The supremum over the set of 1-Lipschitz functions is intractable for most cases, which again suggests the introduction of a learned critic:

$$W_1(p^*(\boldsymbol{x}), q_\theta(\boldsymbol{x})) = \sup_{f : \|f\|_{\text{Lip}} \leq 1} \mathbb{E}_{p^*(\boldsymbol{x})} f(\boldsymbol{x}) - \mathbb{E}_{q_\theta(\boldsymbol{x})} f(\boldsymbol{x}) \tag{26.30}$$

$$\geq \max_{\boldsymbol{\phi} : \|D_{\boldsymbol{\phi}}\|_{\text{Lip}} \leq 1} \mathbb{E}_{p^*(\boldsymbol{x})} D_{\boldsymbol{\phi}}(\boldsymbol{x}) - \mathbb{E}_{q_\theta(\boldsymbol{x})} D_{\boldsymbol{\phi}}(\boldsymbol{x}), \tag{26.31}$$

where the critic $D_{\boldsymbol{\phi}}$ has to be regularized to be 1-Lipschitz (various techniques for Lipschitz regularization via gradient penalties or spectral normalization methods have been used [ACB17; Gul+17]). As was the case with f-divergences, we replace an intractable quantity which requires a supremum over a class of functions with a bound obtained using a subset of this function class, a subset which can be modeled using neural networks.



(a) Optimal Wasserstein critic. *(b) Optimal MMD critic.*

Figure 26.3: Optimal critics in Integral Probability Metrics (IPMs). Generated by ipm_ divergences.ipynb

To train a generative model, we again introduce a min max game:

$$\min_{\boldsymbol{\theta}} W_1(p^*(\boldsymbol{x}), q_\theta(\boldsymbol{x})) \geq \min_{\boldsymbol{\theta}} \max_{\boldsymbol{\phi}:\|D_{\boldsymbol{\phi}}\|_{\text{Lip}}\leq 1} \mathbb{E}_{p^*(\boldsymbol{x})} D_{\boldsymbol{\phi}}(\boldsymbol{x}) - \mathbb{E}_{q_\theta(\boldsymbol{x})} D_{\boldsymbol{\phi}}(\boldsymbol{x}) \tag{26.32}$$

$$= \min_{\boldsymbol{\theta}} \max_{\boldsymbol{\phi}:\|D_{\boldsymbol{\phi}}\|_{\text{Lip}}\leq 1} \mathbb{E}_{p^*(\boldsymbol{x})} D_{\boldsymbol{\phi}}(\boldsymbol{x}) - \mathbb{E}_{q(\boldsymbol{z})} D_{\boldsymbol{\phi}}(G_{\boldsymbol{\theta}}(\boldsymbol{z})) \tag{26.33}$$

This leads to the popular WassersteinGAN [ACB17].

If we replace the choice of function family $\mathcal{F}$ to that of functions in an RKHS (Section 18.3.7.1) with norm one, we obtain the **maximum mean discrepancy** (**MMD**) discussed in Section 2.7.3:

$$\text{MMD}(p^*(\boldsymbol{x}), q_\theta(\boldsymbol{x})) = \sup_{f:\|f\|_{RKHS}=1} \mathbb{E}_{p^*(\boldsymbol{x})} f(\boldsymbol{x}) - \mathbb{E}_{q_\theta(\boldsymbol{x})} f(\boldsymbol{x}). \tag{26.34}$$

We show an example of an MMD critic in Figure 26.3b. It is often more convenient to use the square MMD loss [LSZ15; DRG15], which can be evaluated using the kernel $\mathcal{K}$ (Section 18.3.7.1):

$$\text{MMD}^2(p^*, q_\theta) = \mathbb{E}_{p^*(\boldsymbol{x})}\mathbb{E}_{p^*(\boldsymbol{x}')}\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') - 2\mathbb{E}_{p^*(\boldsymbol{x})}\mathbb{E}_{q_\theta(\boldsymbol{y})}\mathcal{K}(\boldsymbol{x}, \boldsymbol{y}) + \mathbb{E}_{q_\theta(\boldsymbol{y})}\mathbb{E}_{q_\theta(\boldsymbol{y}')}\mathcal{K}(\boldsymbol{y}, \boldsymbol{y}') \tag{26.35}$$

$$= \mathbb{E}_{p^*(\boldsymbol{x})}\mathbb{E}_{p^*(\boldsymbol{x}')}\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}') - 2\mathbb{E}_{p^*(\boldsymbol{x})}\mathbb{E}_{q(\boldsymbol{z})}\mathcal{K}(\boldsymbol{x}, G_{\boldsymbol{\theta}}(\boldsymbol{z})) + \mathbb{E}_{q(\boldsymbol{z})}\mathbb{E}_{q(\boldsymbol{z}')}\mathcal{K}(G_{\boldsymbol{\theta}}(\boldsymbol{z}), G_{\boldsymbol{\theta}}(\boldsymbol{z}')) \tag{26.36}$$

The MMD can be directly used to learn a generative model, often called a generative matching network [LSZ15]:

$$\min_{\boldsymbol{\theta}} \text{MMD}^2(p^*, q_\theta) \tag{26.37}$$

The choice of kernel is important. Using a fixed or predefined kernel such as a radial basis function (RBF) kernel might not be appropriate for all data modalities, such as high dimensional images. Thus we are looking for a way to learn a feature function ζ such that $\mathcal{K}(\zeta(\boldsymbol{x}), \zeta(\boldsymbol{x}'))$ is a valid kernel; luckily, we can use that for any characteristic kernel $\mathcal{K}(\boldsymbol{x}, \boldsymbol{x}')$ and injective function ζ, $\mathcal{K}(\zeta(\boldsymbol{x}), \zeta(\boldsymbol{x}')))$ is also a characteristic kernel. While this tells us that we can use feature functions in the MMD objective, it does not tell us how to learn the features. In order to ensure that the learned features are sensitive to differences between the data distribution $p^*(\boldsymbol{x})$ and the model distribution $q_\theta(\boldsymbol{x})$, the

kernel parameters are trained to *maximize* the square MMD. This again casts the problem into a familiar min max objective by learning the projection ζ with parameters $\boldsymbol{\phi}$ [Li+17b]:

$$\min_{\boldsymbol{\theta}} \mathrm{MMD}_{\zeta}{}^2(p_{\mathcal{D}}, q_\theta) \tag{26.38}$$

$$= \min_{\boldsymbol{\theta}} \max_{\boldsymbol{\phi}} \mathbb{E}_{p^*(\boldsymbol{x})}\mathbb{E}_{p^*(\boldsymbol{x}')}\mathcal{K}(\zeta_{\boldsymbol{\phi}}(\boldsymbol{x}), \zeta_{\boldsymbol{\phi}}(\boldsymbol{x}')) - 2\mathbb{E}_{p^*(\boldsymbol{x})}\mathbb{E}_{q_\theta(\boldsymbol{y})}\mathcal{K}(\zeta_{\boldsymbol{\phi}}(\boldsymbol{x}), \zeta_{\boldsymbol{\phi}}(\boldsymbol{y})) + \mathbb{E}_{q_\theta(\boldsymbol{y})}\mathbb{E}_{q_\theta(\boldsymbol{y}')}\mathcal{K}(\zeta_{\boldsymbol{\phi}}(\boldsymbol{y}), \zeta_{\boldsymbol{\phi}}(\boldsymbol{y}')) \tag{26.39}$$

where $\zeta_{\boldsymbol{\phi}}$ is regularized to be injective, though this is sometimes relaxed [Bin+18]. Unlike the Wasserstein distance and f-divergences, Equation (26.39) can be estimated using Monte Carlo estimation, without requiring a lower bound on the original objective.

26.2.5 Moment matching

More broadly than distances defined by integral probability metrics, for a set of test statistics s, one can define a **moment matching** criteria [Pea36], also known as the method of moments:

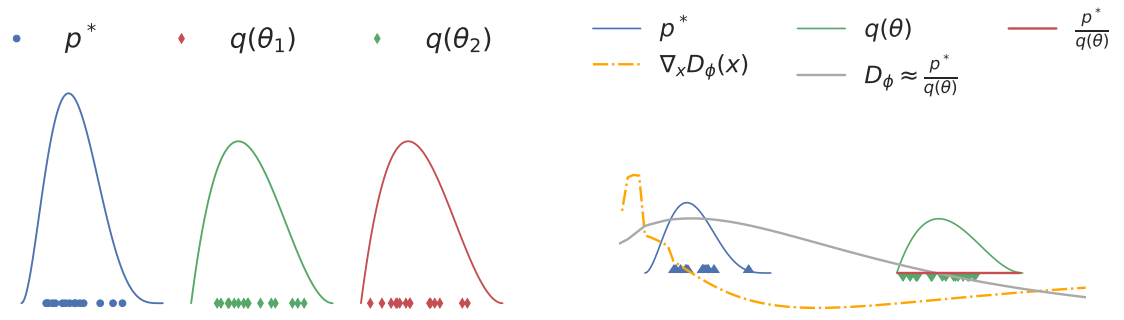
$$\min_{\boldsymbol{\theta}} \left\| \mathbb{E}_{p^*(\boldsymbol{x})} s(\boldsymbol{x}) - \mathbb{E}_{q_\theta(\boldsymbol{x})} s(\boldsymbol{x}) \right\|_2^2 \tag{26.40}$$

where $m(\boldsymbol{\theta}) = \mathbb{E}_{q_\theta(\boldsymbol{x})} s(\boldsymbol{x})$ is the *moment function*. The choice of statistic $s(\boldsymbol{x})$ is crucial, since as with distributional divergences and distances, we would like to ensure that if the objective is minimized and reaches the minimal value 0, the two distributions are the same $p^*(\boldsymbol{x}) = q_\theta(\boldsymbol{x})$. To see that not all functions s satisfy this requirement consider the function $s(\boldsymbol{x}) = \boldsymbol{x}$: simply matching the means of two distributions is not sufficient to match higher moments (such as variance). For likelihood based models the score function $s(\boldsymbol{x}) = \log q_\theta(\boldsymbol{x})$ satisfies the above requirement and leads to a consistent estimator [Vaa00], but this choice of s is not available for implicit generative models.

This motivates the search for other approaches of integrating the method of moments for implicit models. The MMD can be seen as a moment matching criteria, by matching the means of the two distributions after lifting the data into the feature space of an RHKS. But moment matching can go beyond integral probability metrics: Ravuri et al. [Rav+18] show that one can *learn* useful moments by using s as the set of features containing the gradients of a trained discriminator classifier $D_{\boldsymbol{\phi}}$ together with the features of the learned critic: $s_{\boldsymbol{\phi}}(\boldsymbol{x}) = [\nabla_{\boldsymbol{\phi}} D_{\boldsymbol{\phi}}(\boldsymbol{x}), h_1(\boldsymbol{x}), \ldots, h_n(\boldsymbol{x})]$ where $h_1(\boldsymbol{x}), \ldots, h_n(\boldsymbol{x})$ are the hidden activations of the learned critic. Both features and gradients are needed: the gradients $\nabla_{\boldsymbol{\phi}} D_{\boldsymbol{\phi}}(\boldsymbol{x})$ are required to ensure the estimator for the parameters $\boldsymbol{\theta}$ is consistent, since the number of moments $s(\boldsymbol{x})$ needs to be larger than the number of parameters $\boldsymbol{\theta}$, which will be true if the critic will have more parameters than the model; the features $h_i(\boldsymbol{x})$ are added since they have been shown empirically to improve performance, thus showcasing the importance of the choice of test statistics s used to train implicit models.

26.2.6 On density ratios and differences

We have seen how density ratios (Sections 26.2.2 and 26.2.3) and density differences (Section 26.2.4) can be used to define training objectives for implicit generative models. We now explore some of the distinctions between using ratios and differences for learning by comparison, as well as explore the



(a) Failure of the KL divergence to distinguish between distributions with non-overlapping support: $D_{\mathbb{KL}}\left(p^* \parallel q_{\theta_1}\right) = D_{\mathbb{KL}}\left(p^* \parallel q_{\theta_2}\right) = \infty$, despite q_{θ_2} being closer to p^* than q_{θ_1}.

(b) The density ratio $\frac{p^*}{q_\theta}$ used by the KL divergence and a smooth estimate given by an MLP, together with the gradient it provides with respect to the input variable.

Figure 26.4: The KL divergence cannot provide learning signal for distributions without overlapping support (left), while the smooth approximation given by a learned decision surface like an MLP can (right). Generated by ipm_ divergences.ipynb

effects of using approximations to these objectives using function classes such as neural networks has on these distinctions.

One often stated downside of using divergences that rely on density ratios (such as f-divergences) is their poor behavior when the distributions p^* and q_θ do not have overlapping support. For non-overlapping support, the density ratio $\frac{p^*}{q_\theta}$ will be ∞ in the parts of the space where $p^*(\boldsymbol{x}) > 0$ but $q_\theta(\boldsymbol{x}) = 0$, and 0 otherwise. In that case, the $D_{\mathbb{KL}}\left(p^* \parallel q_\theta\right) = \infty$ and the $JSD(p^*, q_\theta) = \log 2$, regardless of the value of $\boldsymbol{\theta}$. Thus *f-divergences cannot distinguish between different model distributions when they do not have overlapping support with the data distribution*, as visualized in Figure 26.4a. This is in contrast with difference based methods such as IPMs such as the Wasserstein distance and the MMD, which have smoothness requirements built in the *definition* of the method, by constraining the norm of the critic (Equations (26.29) and (26.34)). We can see the effect of these constraints in Figure 26.3: both the Wasserstein distance and the MMD provide useful signal in the case of distributions with non-overlapping support.

While the *definition* of f-divergences relies on density ratios (Equation (26.21)), we have seen that to train implicit generative models we use approximations to those divergences obtained using a parametric critic $D_{\boldsymbol{\phi}}$. If the function family of the critic used to approximate the divergence (via the bound or class probability estimation) contains only smooth functions, it will not be able to model the sharp true density ratio, which jumps from 0 to ∞, but it can provide a smooth approximation. We show an example in Figure 26.4b, where we show the density ratio for two distributions without overlapping support and an approximation provided by an MLP trained to approximate the KL divergence using Equation 26.25. Here, the smooth decision surface provided by the MLP can be used to train a generative model while the underlying KL divergence cannot be; the learned MLP provides the gradient signal on how to move distribution mass to areas with more density under the data distribution, while the KL divergence provides a zero gradient almost everywhere in the

space. This ability of approximations to f-divergences to overcome non-overlapping support issues is a desirable property of generative modeling training criteria, as it allows models to learn the data distribution regardless of initialization [Fed+18]. Thus while the case of non-overlapping support provides an important theoretical difference between IPMs and f-divergences, it is less significant in practice since bounds on f-divergences or class probability estimation are used with smooth critics to approximate the underlying divergence.

Some density ratio and density difference based approaches also share commonalities: bounds are used both for f-divergences (variational bounds in Equation 26.25) and for the Wasserstein distance (Equation (26.31)). These bounds to distributional divergence and distances have their own set of challenges: since the generator minimizes a lower bound of the underlying divergence or distance, minimizing this objective provides no guarantees that the divergence will decrease in training. To see this, we can look at Equation 26.26: its RHS can get arbitrarily low without decreasing the LHS, the divergence we are interested in minimizing; this is unlike variational *upper* bound on the KL divergence used to train variational autoencoders Chapter 21.

26.3 Generative adversarial networks

We have looked at different learning principles that do not require the use of explicit likelihoods, and thus can be used to train implicit models. These learning principles specify training criteria, but do not tell us how to *train* models or parameterize models. To answer these questions, we now look at algorithms for training implicit models, where the models (both the discriminator and generator) are deep neural networks; this leads us to generative adversarial networks (GANs). We cover how to turn learning principles into loss functions for training GANs (Section 26.3.1); how to train models using gradient descent (Section 26.3.2); how to improve GAN optimization (Section 26.3.4) and how to assess GAN convergence (Section 26.3.5).

26.3.1 From learning principles to loss functions

In Section 26.2 we discussed learning principles for implicit generative models: class probability estimation, bounds on f-divergences, integral probability metrics and moment matching. These principles can be used to formulate loss functions to train the model parameters $\boldsymbol{\theta}$ and the critic parameters $\boldsymbol{\phi}$. Many of these objectives use **zero-sum losses** via a **min-max** formulation: the generator's goal is to minimize the same function the discriminator is maximizing. We can formalize this as:

$$\min \max V(\boldsymbol{\phi}, \boldsymbol{\theta}) \tag{26.41}$$

As an example, we recover the original GAN with the Bernoulli log-loss (Equation (26.19)) when

$$V(\boldsymbol{\phi}, \boldsymbol{\theta}) = \frac{1}{2}\mathbb{E}_{p^*(\boldsymbol{x})}[\log D_{\boldsymbol{\phi}}(\boldsymbol{x})] + \frac{1}{2}\mathbb{E}_{q_\theta(\boldsymbol{x})}[\log(1 - D_{\boldsymbol{\phi}}(\boldsymbol{x}))]. \tag{26.42}$$

The reason most of the learning principles we have discussed lead to zero-sum losses is due to their underlying structure: the critic maximizes a quantity in order to approximate a divergence or distance — such as an f-divergence or Integral Probability Metric — and the model minimizes this approximation to the divergence or distance. That need not be the case, however. Intuitively,

the discriminator training criteria needs to ensure that the discriminator can distinguish between data and model samples, while the generator loss function needs to ensure that model samples are indistinguishable from data according to the discriminator.

To construct a GAN that is not zero-sum, consider the zero-sum criteria in the original GAN (Equation 26.42), induced by the Bernoulli scoring rule. The discriminator tries to distinguish between data and model samples by classifying the data as real (label 1) and samples as fake (label 0), while the goal of the generator is to minimize the probability that the discriminator classifies its samples as fake: $\min_{\boldsymbol{\theta}} \mathbb{E}_{q_{\boldsymbol{\theta}}(\boldsymbol{x})} \log(1 - D_{\boldsymbol{\phi}}(\boldsymbol{x}))$. An equally intuitive goal for the generator is to maximize the probability that the discriminator classifies its samples as real. While the difference might seem subtle, this loss, known as the "nonsaturating loss" [Goo+14], defined as $\mathbb{E}_{q_{\boldsymbol{\theta}}(\boldsymbol{x})} - \log D_{\boldsymbol{\phi}}(\boldsymbol{x})$, enjoys better gradient properties early in training, as shown in Figure 26.5: the non-saturating loss provides a stronger learning signal (via the gradient) when the generator is performing poorly, and the discriminator can easily distinguish its samples from data, i.e., $D(G(\boldsymbol{z}))$ is low; more on the gradients properties the saturating and non-saturating losses can be found in [AB17; Fed+18].

There exist many other GAN losses which are not zero-sum, including formulations of LS-GAN [Mao+17], GANs trained using the hinge loss [LY17], and RelativisticGANs [JM18]. We can thus generally write a GAN formulation as follows:

$$\min_{\boldsymbol{\phi}} L_D(\boldsymbol{\phi}, \boldsymbol{\theta}); \qquad \min_{\boldsymbol{\theta}} L_G(\boldsymbol{\phi}, \boldsymbol{\theta}). \tag{26.43}$$

We recover the zero-sum formulations if $-L_D(\boldsymbol{\phi}, \boldsymbol{\theta}) = L_G(\boldsymbol{\phi}, \boldsymbol{\theta}) = V(\boldsymbol{\phi}, \boldsymbol{\theta})$. Despite departing from the zero-sum structure, the nested form of the optimization remains in the general formulation, as we will discuss in Section 26.3.2.

The loss functions for the discriminator and generator, L_D and L_G respectively, follow the general form in Equation 26.5, which allows them to be used to efficiently train implicit generative models. The majority of loss functions considered here can thus be written as follows:

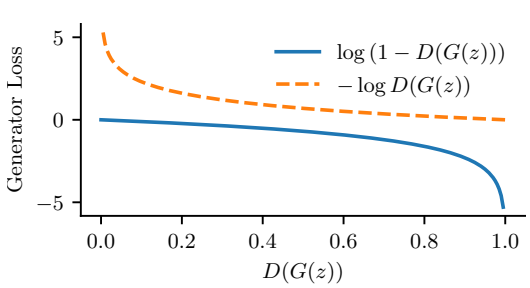
$$L_D(\boldsymbol{\phi}, \boldsymbol{\theta}) = \mathbb{E}_{p^*(\boldsymbol{x})} g(D_{\boldsymbol{\phi}}(\boldsymbol{x})) + \mathbb{E}_{q_{\boldsymbol{\theta}}(\boldsymbol{x})} h(D_{\boldsymbol{\phi}}(\boldsymbol{x})) = \mathbb{E}_{p^*(\boldsymbol{x})} g(D_{\boldsymbol{\phi}}(\boldsymbol{x})) + \mathbb{E}_{q(\boldsymbol{z})} h(D_{\boldsymbol{\phi}}(G_{\boldsymbol{\theta}}(\boldsymbol{z}))) \tag{26.44}$$

$$L_G(\boldsymbol{\phi}, \boldsymbol{\theta}) = \mathbb{E}_{q_{\boldsymbol{\theta}}(\boldsymbol{x})} l(D_{\boldsymbol{\phi}}(\boldsymbol{x})) = \mathbb{E}_{q(\boldsymbol{z})} l(D_{\boldsymbol{\phi}}(G_{\boldsymbol{\theta}}(\boldsymbol{z})) \tag{26.45}$$

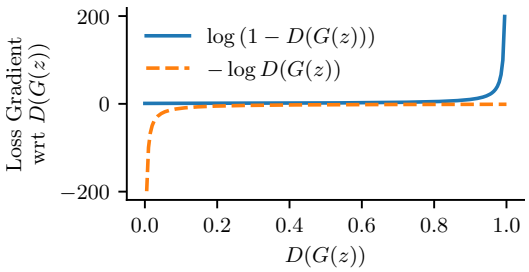
where g, h, $l : \mathbb{R} \to \mathbb{R}$. We recover the original GAN for $g(t) = -\log t$, $h(t) = -\log(1-t)$ and $l(t) = \log(1-t)$; the non-saturating loss for $g(t) = -\log t$, $h(t) = -\log(1-t)$ and $l(t) = -\log(t)$; the Wasserstein distance formulation for $g(t) = t$, $h(t) = -t$ and $l(t) = t$; for f-divergences $g(t) = t$, $h(t) = -f^\dagger(t)$ and $l(t) = f^\dagger(t)$.

26.3.2 Gradient descent

GANs employ the learning principles discussed above in conjunction with gradient based learning for the parameters of the discriminator and generator. We assume a general formulation with a discriminator loss function $L_D(\boldsymbol{\phi}, \boldsymbol{\theta})$ and a generator loss function $L_G(\boldsymbol{\phi}, \boldsymbol{\theta})$. Since the discriminator is often introduced to approximate a distance or divergence $D(p^*, q_\theta)$ (Section 26.2), for the generator to minimize a good approximation of that divergence one should solve the discriminator optimization fully for each generator update. That would entail that for each generator update one would first find the optimal discriminator parameters $\boldsymbol{\phi}^* = \text{argmin}_{\boldsymbol{\phi}} L_D(\boldsymbol{\phi}, \boldsymbol{\theta})$ in order to perform a gradient update given by $\nabla_{\boldsymbol{\theta}} L_G(\boldsymbol{\phi}^*, \boldsymbol{\theta})$. Fully solving the inner optimization problem $\boldsymbol{\phi}^* = \text{argmin}_{\boldsymbol{\phi}} L_D(\boldsymbol{\phi}, \boldsymbol{\theta})$



(a) Generator loss as a function of discriminator score.



(b) The gradients of the generator loss with respect to the discriminator score.

Figure 26.5: Saturating $\log(1 - D(G(z)))$ *vs non-saturating* $-\log D(G(z))$ *loss functions. The non-saturating loss provides stronger gradients when the discriminator is easily detecting that generated samples are fake. Generated by gan_loss_types.ipynb*

for each optimization step of the generator is computationally prohibitive, which motivates the use of alternating updates: performing a few gradient steps to update the discriminator parameters, followed by a generator update. Note that when updating the discriminator, we keep the generator parameters fixed, and when updating the generator, we keep the discriminator parameters fixed. We show a general algorithm for these alternative updates in Algorithm 26.1.

Algorithm 26.1: General GAN training algorithm with alternating updates

Initialize $\boldsymbol{\phi}$, $\boldsymbol{\theta}$
for *each training iteration* **do**
  for K *steps* **do**
    Update the discriminator parameters $\boldsymbol{\phi}$ using the gradient $\nabla_{\boldsymbol{\phi}} L_D(\boldsymbol{\phi}, \boldsymbol{\theta})$;
  Update the generator parameters $\boldsymbol{\theta}$ using the gradient $\nabla_{\boldsymbol{\theta}} L_G(\boldsymbol{\phi}, \boldsymbol{\theta})$
Return $\boldsymbol{\phi}$, $\boldsymbol{\theta}$

We are thus interested in computing $\nabla_{\boldsymbol{\phi}} L_D(\boldsymbol{\phi}, \boldsymbol{\theta})$ and $\nabla_{\boldsymbol{\theta}} L_G(\boldsymbol{\phi}, \boldsymbol{\theta})$. Given the choice of loss functions follows the general form in Equations 26.44 and 26.45 both for the discriminator and generator, we can compute the gradients that can be used for training. To compute the discriminator gradients, we write:

$$\nabla_{\boldsymbol{\phi}} L_D(\boldsymbol{\phi}, \boldsymbol{\theta}) = \nabla_{\boldsymbol{\phi}} \left[\mathbb{E}_{p^*(\boldsymbol{x})} g(D_{\boldsymbol{\phi}}(\boldsymbol{x})) + \mathbb{E}_{q_{\boldsymbol{\theta}}(\boldsymbol{x})} h(D_{\boldsymbol{\phi}}(\boldsymbol{x})) \right] \tag{26.46}$$

$$= \mathbb{E}_{p^*(\boldsymbol{x})} \nabla_{\boldsymbol{\phi}} g(D_{\boldsymbol{\phi}}(\boldsymbol{x})) + \mathbb{E}_{q_{\boldsymbol{\theta}}(\boldsymbol{x})} \nabla_{\boldsymbol{\phi}} h(D_{\boldsymbol{\phi}}(\boldsymbol{x})) \tag{26.47}$$

where $\nabla_{\boldsymbol{\phi}} g(D_{\boldsymbol{\phi}}(\boldsymbol{x}))$ and $\nabla_{\boldsymbol{\phi}} h(D_{\boldsymbol{\phi}}(\boldsymbol{x}))$ can be computed via backpropagation, and each expectation can be estimated using Monte Carlo estimation. For the generator, we would like to compute the gradient:

$$L_G(\boldsymbol{\phi}, \boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \mathbb{E}_{q_{\boldsymbol{\theta}}(\boldsymbol{x})} l(D_{\boldsymbol{\phi}}(\boldsymbol{x})) \tag{26.48}$$

Here we cannot change the order of differentiation and integration since the distribution under the integral depends on the differentiation parameter $\boldsymbol{\theta}$. Instead, we will use that $q_\theta(\boldsymbol{x})$ is the distribution induced by an implicit generative model (also known as the "reparameterization trick", see Section 6.3.5):

$$\nabla_{\boldsymbol{\theta}} L_G(\boldsymbol{\phi}, \boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \mathbb{E}_{q_\theta(\boldsymbol{x})} l(D_{\boldsymbol{\phi}}(\boldsymbol{x})) = \nabla_{\boldsymbol{\theta}} \mathbb{E}_{q(\boldsymbol{z})} l(D_{\boldsymbol{\phi}}(G_{\boldsymbol{\theta}}(\boldsymbol{z}))) = \mathbb{E}_{q(\boldsymbol{z})} \nabla_{\boldsymbol{\theta}} l(D_{\boldsymbol{\phi}}(G_{\boldsymbol{\theta}}(\boldsymbol{z}))) \tag{26.49}$$

and again use Monte Carlo estimation to approximate the gradient using samples from the prior $q(\boldsymbol{z})$. Replacing the choice of loss functions and Monte Carlo estimation in Algorithm 26.1 leads to Algorithm 26.2, which is often used to train GANs.

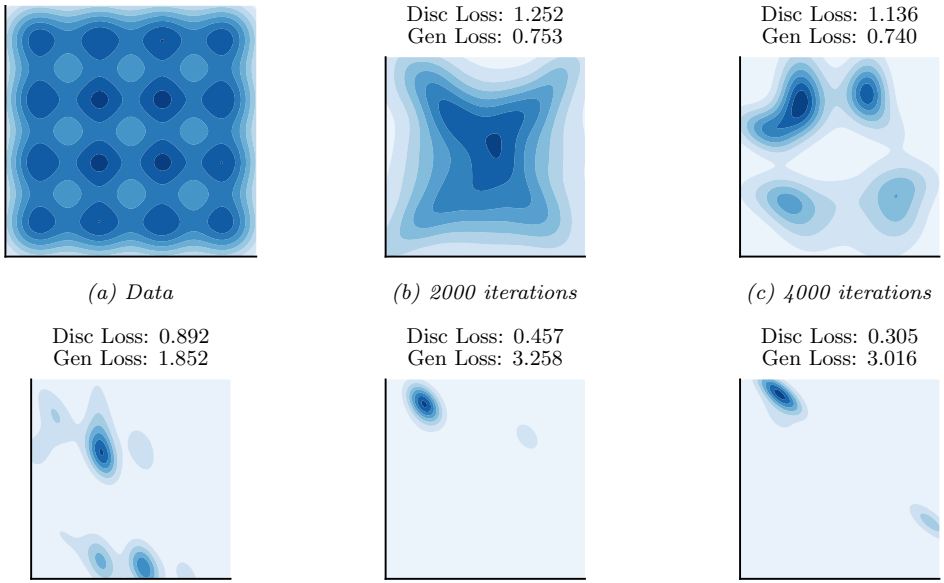
Algorithm 26.2: GAN training algorithm

Initialize $\boldsymbol{\phi}$, $\boldsymbol{\theta}$
for *each training iteration* **do**
  for K *steps* **do**
    Sample minibatch of M noise vectors $\boldsymbol{z}_m \sim q(\boldsymbol{z})$
    Sample minibatch of M examples $\boldsymbol{x}_m \sim p^*(\boldsymbol{x})$
    Update the discriminator by performing stochastic gradient descent using this gradient:
    $\nabla_{\boldsymbol{\phi}} \frac{1}{M} \sum_{m=1}^{M} [g(D_{\boldsymbol{\phi}}(\boldsymbol{x}_m)) + \nabla_{\boldsymbol{\phi}} h(D_{\boldsymbol{\phi}}(G_{\boldsymbol{\theta}}(\boldsymbol{z}_m)))]$.
  Sample minibatch of M noise vectors $\boldsymbol{z}_m \sim q(\boldsymbol{z})$
  Update the generator by performing stochastic gradient descent using this gradient:
  $\nabla_{\boldsymbol{\theta}} \frac{1}{M} \sum_{m=1}^{M} l(D_{\boldsymbol{\phi}}(G_{\boldsymbol{\theta}}(\boldsymbol{z}_m))$.
Return $\boldsymbol{\phi}$, $\boldsymbol{\theta}$

26.3.3 Challenges with GAN training

Due to the adversarial game nature of GANs the optimizing dynamics of GANs are both hard to study in theory, and to stabilize in practice. GANs are known to suffer from **mode collapse**, a phenomenon where the generator converges to a distribution which does not cover not all the modes (peaks) of the data distribution, thus the model underfits the distribution. We show an example in Figure 26.6: while the data is a mixture of Gaussians with 16 modes, the model converges only to a few modes. Alternatively, another problematic behavior is **mode hopping**, where the generator "hops" between generating different modes of the data distribution. An intuitive explanation for this behavior is as follows: if the generator becomes good at generating data from one mode, it will generate more from that mode. If the discriminator cannot learn to distinguish between real and generated data in this mode, the generator has no incentive to expand its support and generate data from other modes. On the other hand, if the discriminator eventually learns to distinguish between the real and generated data inside this mode, the generator can simply move (hop) to a new mode, and this game of cat and mouse can continue.

While mode collapse and mode hopping are often associated with GANs, many improvements have made GAN training more stable, and these behaviors more rare. These improvements include using large batch sizes, increasing the discriminator neural capacity, using discriminator and generator regularization, as well as more complex optimization methods.



(a) Data *(b) 2000 iterations* *(c) 4000 iterations*

(d) 8000 iterations *(e) 12000 iterations* *(f) 14000 iterations*

Figure 26.6: Illustration of mode collapse and mode hopping in GAN training. (a) The dataset, a mixture of 16 Gaussians in 2 dimensions. (b-f) Samples from the model after various amounts of training. Generated by gan_mixture_of_gaussians.ipynb.

26.3.4 Improving GAN optimization

Hyperparameter choices such as the choice of momentum can be crucial when training GANs, with lower momentum values being preferred compared to the usual high momentum used in supervised learning. Algorithms such as Adam [KB14a] provide a great boost in performance [RMC16a]. Many other optimization methods have been successfully applied to GANs, such as those which target variance reduction [Cha+19c]; those which backpropagate through gradient steps, thus ensuring that generator does well against the discriminator *after it has been updated* [Met+16]; or using a local bilinear approximation of the two player game [SA19]. While promising, these advanced optimization methods tend to have a higher computational cost, making them harder to scale to large models or large datasets compared to less efficient optimization methods.

26.3.5 Convergence of GAN training

The challenges with GAN optimization make it hard to quantify when convergence has occurred. In Section 26.2 we saw how global convergence guarantees can be provided under optimality conditions for multiple objectives constructed starting with different distributional divergences and distances: if the discriminator is optimal, the generator is minimizing a distributional divergence or distance between the data and model distribution, and thus under infinite capacity and perfect optimization can learn the data distribution. This type of argument has been used since the original GAN paper [Goo+14]

to connect GANs to standard objectives in generative models, and obtain the associated theoretical guarantees. From a game theory perspective, this type of convergence guarantee provides an existence proof of a global Nash equilibrium for the GAN game, though under strong assumptions. A Nash equilibrium is achieved when both players (the discriminator and generator) would incur a loss if they decide to act by changing their parameters. Consider the original GAN defined by the objective in Equation 26.19; then $q_{\boldsymbol{\theta}} = p^*$ and $D_{\boldsymbol{\phi}}(\boldsymbol{x}) = \frac{p^*(\boldsymbol{x})}{p^*(\boldsymbol{x})+q_\theta(\boldsymbol{x})} = \frac{1}{2}$ is a global Nash equilibrium, since for a given $q_{\boldsymbol{\theta}}$, the ratio $\frac{p^*(\boldsymbol{x})}{p^*(\boldsymbol{x})+q_\theta(\boldsymbol{x})}$is the optimal discriminator (Equation 26.11), and given an optimal discriminator, the data distribution is the optimal generator as it is the minimizer of the Jensen-Shannon divergence (Equation 26.15).

While these global theoretical guarantees provide useful insights about the GAN game, they do not account for optimization challenges that arise with accounting for the optimization trajectories of the two players, or for neural network parameterization since they assume infinite capacity both for the discriminator and generator. In practice GANs do not decrease a distance or divergence at every optimization step [Fed+18] and global guarantees are difficult to obtain when using optimization methods such as gradient descent. Instead, the focus shifts towards local convergence guarantees, such as reaching a local Nash equilibrium. A local Nash equilibrium requires that both players are at a local, not global minimum: a local Nash equilibrium is a stationary point (the gradients of the two loss functions are zero, i.e $\nabla_{\boldsymbol{\phi}} L_D(\boldsymbol{\phi}, \boldsymbol{\theta}) = \mathbf{0}$ and $\nabla_{\boldsymbol{\theta}} L_G(\boldsymbol{\phi}, \boldsymbol{\theta}) = \mathbf{0}$), and the eigenvalues of the Hessian of each player ($\nabla_{\boldsymbol{\phi}} \nabla_{\boldsymbol{\phi}} L_D(\boldsymbol{\phi}, \boldsymbol{\theta})$ and $\nabla_{\boldsymbol{\theta}} \nabla_{\boldsymbol{\theta}} L_G(\boldsymbol{\phi}, \boldsymbol{\theta})$) are non-negative; for a longer discussion on Nash equilibria in continuous games, see [RBS16]. For the general GAN game, it is not guaranteed that a local Nash equilibrium always exists [FO20], and weaker conditions such as stationarity or locally stable stationarity have been studied [Ber+19]; other equilibrium definitions inspired by game theory have also been used [JNJ20; HLC19].

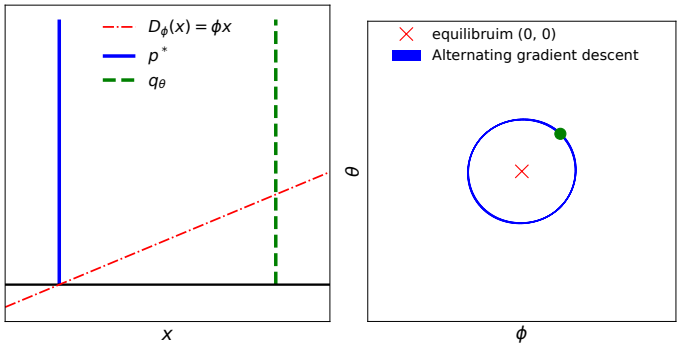
To motivate why convergence analysis is important in the case of GANs, we visualize an example of a GAN that does not converge trained with gradient descent. In DiracGAN [MGN18a] the data distribution $p^*(\boldsymbol{x})$ is the Dirac delta distribution with mass at zero. The generator is modeling a Dirac delta distribution with parameter θ: $G_\theta(z) = \theta$ and the discriminator is a linear function of the input with learned parameter ϕ: $D_\phi(x) = \phi x$. We also assume a GAN formulation where $g = h = -l$ in the general loss functions L_D and L_G defined above, see Equations (26.44) and (26.45). This results in the zero-sum game given by:

$$L_D = \mathbb{E}_{p^*(x)} - l(D_{\boldsymbol{\phi}}(x)) + \mathbb{E}_{q_\theta(x)} - l(D_{\boldsymbol{\phi}}(x)) = -l(0) - l(\theta\phi) \tag{26.50}$$

$$L_G = \mathbb{E}_{p^*(x)} l(D_{\boldsymbol{\phi}}(x)) + \mathbb{E}_{q_\theta(x)} l(D_{\boldsymbol{\phi}}(x)) = +l(0) + l(\theta\phi) \tag{26.51}$$

where l depends on the GAN formulation used ($l(z) = -\log(1 + e^{-z})$ for instance). The unique equilibrium point is $\theta = \phi = 0$. We visualize the DiracGAN problem in Figure 26.7 and show that DiracGANs with alternating gradient descent (Algorithm 26.1) do not reach the equilibrium point, but instead takes a circular trajectory around the equilibrium.

There are two main theoretical approaches taken to understand GAN convergence behavior around an equilibrium: by analyzing either the discrete dynamics of gradient descent, or the underlying continuous dynamics of the game using approaches such as stability analysis. To understand the difference between the two approaches, consider the discrete dynamics defined by gradient descent



(a) The DiracGAN problem. *(b) Alternating gradient descent on DiracGAN.*

Figure 26.7: Visualizing divergence using a simple GAN: DiracGAN. Generated by dirac_gan.ipynb

with learning rates αh and λh, either via alternating updates (as we have seen in Algorithm 26.1):

$$\boldsymbol{\phi}_t = \boldsymbol{\phi}_{t-1} - \alpha h \nabla_{\boldsymbol{\phi}} L_D(\boldsymbol{\phi}_{t-1}, \boldsymbol{\theta}_{t-1}), \tag{26.52}$$

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \lambda h \nabla_{\boldsymbol{\theta}} L_G(\boldsymbol{\phi}_t, \boldsymbol{\theta}_{t-1}) \tag{26.53}$$

or simultaneous updates, where instead of alternating the gradient updates between the two players, they are both updated simultaneously:

$$\boldsymbol{\phi}_t = \boldsymbol{\phi}_{t-1} - \alpha h \nabla_{\boldsymbol{\phi}} L_D(\boldsymbol{\phi}_{t-1}, \boldsymbol{\theta}_{t-1}), \tag{26.54}$$

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \lambda h \nabla_{\boldsymbol{\theta}} L_G(\boldsymbol{\phi}_{t-1}, \boldsymbol{\theta}_{t-1}) \tag{26.55}$$

The above dynamics of gradient descent are obtained using Euler numerical integration from the ODEs that describes the game dynamics of the two players:

$$\dot{\boldsymbol{\phi}} = -\nabla_{\boldsymbol{\phi}} L_D(\boldsymbol{\phi}, \boldsymbol{\theta}), \tag{26.56}$$

$$\dot{\boldsymbol{\theta}} = -\nabla_{\boldsymbol{\theta}} L_G(\boldsymbol{\phi}, \boldsymbol{\theta}) \tag{26.57}$$

One approach to understand the behavior of GANs is to study these underlying ODEs, which, when discretized, result in the gradient descent updates above, rather than directly studying the discrete updates. These ODEs can be used for stability analysis to study the behavior around an equilibrium. This entails finding the eigenvalues of the Jacobian of the game

$$J = \begin{bmatrix} -\nabla_{\boldsymbol{\phi}} \nabla_{\boldsymbol{\phi}} L_D(\boldsymbol{\phi}, \boldsymbol{\theta}) & -\nabla_{\boldsymbol{\theta}} \nabla_{\boldsymbol{\phi}} L_D(\boldsymbol{\phi}, \boldsymbol{\theta}) \\ -\nabla_{\boldsymbol{\phi}} \nabla_{\boldsymbol{\theta}} L_G(\boldsymbol{\phi}, \boldsymbol{\theta}) & -\nabla_{\boldsymbol{\theta}} \nabla_{\boldsymbol{\theta}} L_G(\boldsymbol{\phi}, \boldsymbol{\theta}) \end{bmatrix} \tag{26.58}$$

evaluated at a stationary point (i.e., where $\nabla_{\boldsymbol{\phi}} L_D(\boldsymbol{\phi}, \boldsymbol{\theta}) = 0$ and $\nabla_{\boldsymbol{\theta}} L_G(\boldsymbol{\phi}, \boldsymbol{\theta}) = 0$). If the eigenvalues of the Jacobian all have negative real parts, then the system is asymptotically stable around the equilibrium; if at least one eigenvalue has positive real part, the system is unstable around the

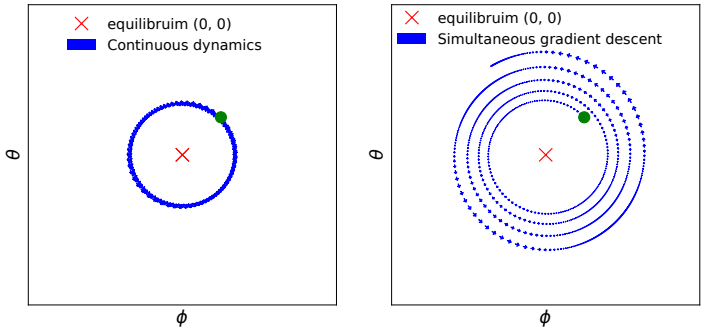


Figure 26.8: Continuous (left) and discrete dynamics (right) take different trajectories in DiracGAN. Generated by dirac_gan.ipynb

equilibrium. For the DiracGAN, the Jacobian evaluated at the equilibrium $\theta = \phi = 0$ is:

$$J = \begin{bmatrix} \nabla_{\boldsymbol{\phi}} \nabla_{\boldsymbol{\phi}} l(\theta\phi) + l(0) & \nabla_{\boldsymbol{\theta}} \nabla_{\boldsymbol{\phi}} l(\theta\phi) + l(0) \\ -\nabla_{\boldsymbol{\phi}} \nabla_{\boldsymbol{\theta}} \left(l(\theta\phi) + l(0) \right) & -\nabla_{\boldsymbol{\theta}} \nabla_{\boldsymbol{\theta}} \left(l(\theta\phi) + l(0) \right) \end{bmatrix} = \begin{bmatrix} 0 & l'(0) \\ -l'(0) & 0 \end{bmatrix} \tag{26.59}$$

where eigenvalues of this Jacobian are $\lambda_{\pm} = \pm i l'(0)$. This is interesting, as the real parts of the eigenvalues are both 0; this result tells us that there is no asymptotic convergence to an equilibrium, but linear convergence could still occur. In this simple case we can reach the conclusion that convergence does not occur as we observe that there is a preserved quantity in this system, as $\theta^2 + \phi^2$ does not change in time (Figure 26.8, left):

$$\frac{d\left(\theta^2 + \phi^2\right)}{dt} = 2\theta \frac{d\theta}{dt} + 2\phi \frac{d\phi}{dt} = -2\theta l'(\theta\phi)\phi + 2\phi l'(\theta\phi)\theta = 0.$$

Using stability analysis to understand the underlying continuous dynamics of GANs around an equilibrium has been used to show that explicit regularization can help convergence [NK17; Bal+18]. Alternatively, one can directly study the updates of simultaneous gradient descent shown in Equations 26.54 and 26.55. Under certain conditions, [MNG17b] prove that GANs trained with simultaneous gradient descent reach a local Nash equilibrium. Their approach relies on assessing the convergence of series of the form $F^k(\boldsymbol{x})$ resulting from the repeated application of gradient descent update of the form $F(\boldsymbol{x}) = \boldsymbol{x} + hG(\boldsymbol{x})$, where h is the learning rate. Since the function F depends on the learning rate h, their convergence results depend on the size of the learning rate, which is not the case for continuous time approaches.

Both continuous and discrete approaches have been useful in understanding and improving GAN training; however, both approaches still leave a gap between our theoretical understanding and the most commonly used algorithms to train GANs in practice, such as alternating gradient descent or more complex optimizers used in practice, like Adam. Far from only providing different proof techniques, these approaches can reach different conclusions about the convergence of a GAN: we show an example in Figure 26.8, where we see that simultaneous gradient descent and the continuous dynamics behave differently when a large enough learning rate is used. In this case, the discretization error — the difference between the behavior of the continuous dynamics in Equations 26.56 and 26.57

and the gradient descent dynamics in Equations 26.54 and 26.55 — makes the analysis of gradient descent using continuous dynamics reach the wrong conclusion about DiracGAN [Ros+21]. This difference in behavior has been a motivator to train GANs with higher order numerical integrators such as RungeKutta4, which to more closely follow the underlying continuous system compared to gradient descent [Qin+20].

While optimization convergence analysis is an indispensable step in understanding GAN training and has led to significant practical improvements, it is worth noting that ensuring converge to an equilibrium does not ensure the model has learned a good fit of the data distribution. The loss landscape determined by the choice of L_D and L_G, as well as the parameterization of the discriminator and generator can lead to equilibria which do not capture the data distribution. The lack of distributional guarantees provided by game equilibria showcases the need to complement convergence analysis with work looking at the effect of gradient based learning in this game setting on the learned distribution.

26.4 Conditional GANs

We have thus far discussed how to use implicit generative models to learn a true unconditional distribution $p^*(\boldsymbol{x})$ from which we only have samples. It is often useful, however, to be able to learn *conditional distributions* of the from $p^*(\boldsymbol{x}|\boldsymbol{y})$. This requires having **paired data**, where each input $\boldsymbol{x}_n$ is paired with a corresponding set of covariates $\boldsymbol{y}_n$, such as a class label, or a set of attributes or words, so $\mathcal{D} = \{(\boldsymbol{x}_n, \boldsymbol{y}_n) : n = 1 : N\}$, as in standard supervised learning. The conditioning variable can be discrete, like a class label, or continuous, such as an embedding which encodes other information. Conditional generative models are appealing since we can specify that we want the generated sample to be associated with conditioning information y, making them very amenable to real world applications, as we discuss in Section 26.7.

To be able to learn implicit conditional distributions $q_\theta(\boldsymbol{x}|\boldsymbol{y})$, we require datasets that specify the conditioning information associated with data, and we have to adapt the model architectures and loss functions. In the GAN case, changing the loss function for the generative model can be done by changing the critic, since the critic is part of the loss function of the generator; it is important for the critic to provide learning signal accounting for conditioning information, by penalizing a generator which provides realistic samples but which ignore the provided conditioning.

If we do not change the form of the min-max game, but provide the conditioning information to the two players, a **conditional GAN** can be created from the original GAN game [MO14]:

$$\min_{\boldsymbol{\theta}} \max_{\boldsymbol{\phi}} \frac{1}{2}\mathbb{E}_{p(\boldsymbol{y})}\mathbb{E}_{p^*(\boldsymbol{x}|\boldsymbol{y})}[\log D_{\boldsymbol{\phi}}(\boldsymbol{x}, \boldsymbol{y})] + \frac{1}{2}\mathbb{E}_{p(\boldsymbol{y})}\mathbb{E}_{q_\theta(\boldsymbol{x}|\boldsymbol{y})}[\log(1 - D_{\boldsymbol{\phi}}(\boldsymbol{x}, \boldsymbol{y}))] \tag{26.60}$$

In the case of implicit latent variable models, the embedding information becomes an additional input to the generator, together with the latent variable $\boldsymbol{z}$:

$$\min_{\boldsymbol{\theta}} \max_{\boldsymbol{\phi}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}) = \frac{1}{2}\mathbb{E}_{p(\boldsymbol{y})}\mathbb{E}_{p^*(\boldsymbol{x}|\boldsymbol{y})}[\log D_{\boldsymbol{\phi}}(\boldsymbol{x}, \boldsymbol{y})] + \frac{1}{2}\mathbb{E}_{p(\boldsymbol{y})}\mathbb{E}_{q(\boldsymbol{z})}[\log(1 - D_{\boldsymbol{\phi}}(\mathcal{G}_{\boldsymbol{\theta}}(\boldsymbol{z}, \boldsymbol{y}), \boldsymbol{y}))] \tag{26.61}$$

For discrete conditioning information such as labels, one can also add a new loss function, by training a critic which does not only learn to distinguish between real and fake data, but learns to classify both data and generated samples as pertaining to one of the K classes provided in the

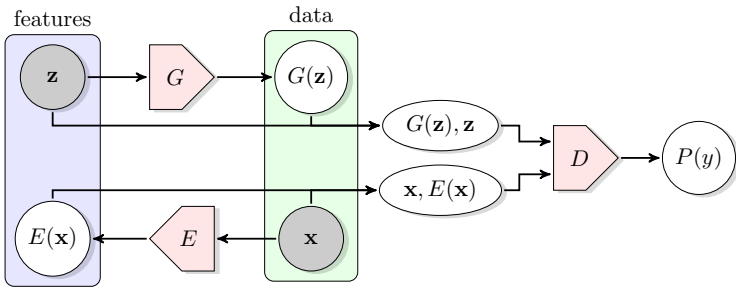


Figure 26.9: Learning an implicit posterior using an adversarial approach, as done in BiGAN. From Figure 1 of [DKD16]. Used with kind permission of Jeff Donahue.

dataset [OOS17]:

$$\mathcal{L}_c(\boldsymbol{\theta}, \boldsymbol{\phi}) = -\left[\frac{1}{2}\mathbb{E}_{p(\boldsymbol{y})}\mathbb{E}_{p^*(\boldsymbol{x}|\boldsymbol{y})}[\log D_{\boldsymbol{\phi}}(\boldsymbol{y}|\boldsymbol{x})] + \frac{1}{2}\mathbb{E}_{p(\boldsymbol{y})}\mathbb{E}_{q_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{y})}[\log(D_{\boldsymbol{\phi}}(\boldsymbol{y}|\boldsymbol{x}))]\right] \tag{26.62}$$

Note that while we could have two critics, one unsupervised critic and one supervised which maximizes the equation above, in practice the same critic is used, to aid shaping the features used in both decision surfaces. Unlike the adversarial nature of the unsupervised game, it is in the interest of both players to minmize the classification loss $\mathcal{L}_c$. Thus together with the adversarial dynamics provided by $\mathcal{L}$, the two players are trained as follows:

$$\max_{\boldsymbol{\phi}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}) - \mathcal{L}_c(\boldsymbol{\theta}, \boldsymbol{\phi}) \qquad \min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}) + \mathcal{L}_c(\boldsymbol{\theta}, \boldsymbol{\phi}) \tag{26.63}$$

In the case of conditional latent variable models, the latent variable controls the sample variability *inside* the mode specified by the conditioning information. In early conditional GANs, the conditioning information was provided as additional input to the discriminator and generator, for example by concatenating the conditioning information to the latent variable $\boldsymbol{z}$ in the case of the generator; it has been since observed that it is important to provide the conditioning information at various layers of the model, both for the generator and the discriminator [DV+17; DSK16] or use a projection discriminator [MK18].

26.5 Inference with GANs

Unlike other latent variable models such as variational autoencoders, GANs do not define an inference procedure associated with the generative model. To deploy the principles behind GANs to find a posterior distribution $p(\boldsymbol{z}|\boldsymbol{x})$, multiple approaches have been taken, from combining GANs and variational autoencoders via hybrid methods [MNG17a; Sri+17; Lar+16; Mak+15b] to constructing inference methods catered to implicit variable models [Dum+16; DKD16; DS19]. An overview of these methods can be found in [Hus17b].

GAN based methods which perform inference and learn **implicit posterior distribution** $p(\boldsymbol{z}|\boldsymbol{x})$ introduce changes to the GAN algorithm to do so. An example of such a method is **BiGAN** (bidirectional GAN) [DKD16] or **ALI** (adversarialy learned inference) [Dum+16], which trains an

implicit parameterized encoder $E_{\boldsymbol{\zeta}}$ to map input $\boldsymbol{x}$ to latent variables $\boldsymbol{z}$. To ensure consistency between the encoder $E_{\boldsymbol{\zeta}}$ and the generator $G_{\boldsymbol{\theta}}$, an adversarial approach is introduced with a discriminator $D_{\boldsymbol{\phi}}$ learning to distinguish between pairs of data and latent samples: $D_{\boldsymbol{\phi}}$ learns to consider pairs $(\boldsymbol{x}, E_{\boldsymbol{\zeta}}(\boldsymbol{x}))$ with $\boldsymbol{x} \sim p^*$ as real, while $(G_{\boldsymbol{\theta}}(\boldsymbol{z}), \boldsymbol{z})$ with $\boldsymbol{z} \sim q(\boldsymbol{z})$ is considered fake. This approach, shown in Figure 26.9, ensures that the joint distributions are matched, and thus the marginal distribution $q_\theta(\boldsymbol{x})$ given by $G_{\boldsymbol{\theta}}$ should learn $p^*(\boldsymbol{x})$, while the conditional distribution $p_{\boldsymbol{\zeta}}(\boldsymbol{z}|\boldsymbol{x})$ given by $E_{\boldsymbol{\zeta}}$ should learn $q_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}) = \frac{q_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z})}{q_{\boldsymbol{\theta}}(\boldsymbol{x})} \propto q_\theta(\boldsymbol{x}|\boldsymbol{z})q(\boldsymbol{z})$. This joint GAN loss can be used both to train the generator $G_{\boldsymbol{\theta}}$ and the encoder $E_{\boldsymbol{\zeta}}$, without requiring a reconstruction loss common in other inference methods. While not using a reconstruction loss, this objective retains the property that under global optimality conditions the encoder and decoder are inverses of each other: $E_{\boldsymbol{\theta}}(G_{\boldsymbol{\zeta}}(\boldsymbol{z})) = \boldsymbol{z}$ and $G_{\boldsymbol{\zeta}}(E_{\boldsymbol{\theta}}(\boldsymbol{x})) = \boldsymbol{x}$. (See also Section 21.2.4 for a discussion of how VAEs learn to ensure $p^*(\boldsymbol{x})p_{\boldsymbol{\zeta}}(\boldsymbol{z}|\boldsymbol{x})$ matches $p(\boldsymbol{z})p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$ using an explicit model of the data.)

26.6 Neural architectures in GANs

We have so far discussed the learning principles, algorithms, and optimization methods that can be used to train implicit generative models parameterized by deep neural networks. We have not discussed, however, the importance of the choice of neural network architectures for the model and the critic, choices which have fueled the progress in GAN generation since their conception. We will look at a few case studies which show the importance of information about data modalities into the critic and the generator (Section 26.6.1), employing the right inductive biases (Section 26.6.2), incorporating attention in GAN models (Section 26.6.3), progressive generation (Section 26.6.4), regularization (Section 26.6.5), and using large scale architectures (Section 26.6.6).

26.6.1 The importance of discriminator architectures

Since the discriminator or critic is rarely optimal — either due to the use of alternating gradient descent or the lack of capacity of the neural discriminator — GANs do not perform distance or divergence minimization in practice. Instead, the critic acts as part of a **learned loss function** for the model (the generator). Every time the critic is updated, the loss function for the generative model changes; this is in stark contrast with divergence minimization such maximum likelihood estimation, where the loss function stays the same throughout the training of the model. Just as learning features of data instead of handcrafting them is a reason for the success of deep learning methods, learning loss functions advanced the state of the art of generative modeling. Critics that take data modalities into account — such as convolutional critics for images and recurrent critics for sequential data such as text or audio — become part of data modality dependent loss functions. This in turn provides modality-specific learning signal to the model, for example by penalizing blurry images and encouraging sharp edges, which is achieved due to the convolutional parameterization of the critic. Even within the same data modality, changes to critic architectures and regularization have been one of the main drivers in obtaining better GANs, since they affect the generator's loss function, and thus also the *gradients of the generator* and have a strong effect on optimization.

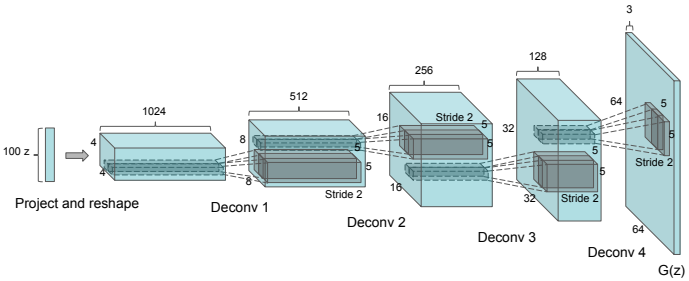


Figure 26.10: DCGAN convolutional generator. From Figure 1 of [RMC15]. Used with kind permission of Alec Radford.

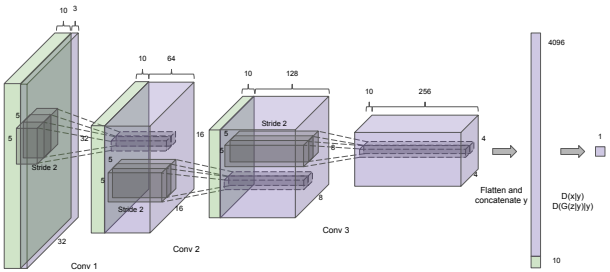


Figure 26.11: DCGAN convolutional discrimiantor. From Figure 1 of [RMC15]. Used with kind permission of Alec Radford.

26.6.2 Architectural inductive biases

While the original GAN paper used convolutions only sparingly, deep convolutional GAN (**DCGAN**) [RMC15] performed an extensive study on what architectures are most useful for GAN training, resulting in a set of useful guidelines that led to a substantial boost in performance. Without changing the learning principles behind GANs, DCGAN was able to obtain better results on image data by using convolutional generators (Figure 26.10) and critics, using BatchNormalization for both the generator and critic, replacing pooling layers with strided convolutions, using ReLU activation networks in the generator, and LeakyReLU activations in the discriminator. Many of these principles are still in use today, for larger architectures and with various loss functions. Since DCGAN, residual convolutional layers have become a key staple of both models and critics for image data [Gul+17], and recurrent architectures are used for sequence data such as text [SSG18b; Md+19].

26.6.3 Attention in GANs

Attention mechanisms are explained in detail in Section 16.2.7. In this section, we discuss how to use them for both the GAN generator and discriminator; this is called the self attention GAN or **SAGAN** model [Zha+19c]. The advantage of self attention is that it ensures that both discriminator and generator have access to a *global* view of other units of the same layer, unlike convolutional

Figure 26.12: Attention queries used by a SAGAN model, showcasing the global span of attention. Each row first shows the input image and a set of color coded query locations in the image. The subsequent images show the attention maps corresponding to each query location in the first image, with the query color coded location being shown, and arrows from it to the attention map are used to highlight the most attended regions. From Figure 1 of [Zha+19c]. Used with kind permission of Han Zhang.

layers. This is illustrated in Figure 26.12, which visualizes the global span of attention: query points can attend to various other areas in the image.

The self-attention mechanism for convolutional features reshaped to $\boldsymbol{h} \in \mathbb{R}^{C \times N}$ is defined by $\boldsymbol{f} = W_f \boldsymbol{h}$, $\boldsymbol{g} = W_g \boldsymbol{h}$, $\mathbf{S} = \boldsymbol{f}^T \boldsymbol{g}$, where $W_f \in \mathbb{R}^{C' \times C}, W_g \in \mathbb{R}^{C' \times C}$, where $C' \leq C$ is a hyperparameter. From $\mathbf{S} \in \mathbb{R}^{N \times N}$, a probability row matrix β is obtained by applying the softmax operator for each row, which is then used to *attend* to a linear transformation of the features $\boldsymbol{o} = W_o(W_h \boldsymbol{h})\beta^T \in R^{C \times N}$, using learned operators $W_h \in \mathbb{R}^{C' \times C}, W_o \in \mathbb{R}^{C \times C'}$. An output is then created by $\boldsymbol{y} = \gamma \boldsymbol{o} + \boldsymbol{h}$, where $\gamma \in \mathbb{R}$ is a learned parameter.

Beyond providing global signal to the players, it is worth noting the flexibility of the self attention mechanism. The learned parameter γ ensures that the model can decide not to use the attention layer, and thus adding self attention does not restrict the set of possible models an architecture can learn. Moreover, self attention significantly increases the number of parameters of the model (each attention layer introduced 4 learned matrices $\mathbf{W}_f, \mathbf{W}_g, \mathbf{W}_h, \mathbf{W}_o$), an approach that has been observed as a fruitful way to improve GAN training.

26.6.4 Progressive generation

One of the first successful approaches to generating higher resolution, color images from a GAN is via an *iterative* process, by first generating a lower dimensional sample, and then using that as conditioning information to generate a higher dimensional sample, and repeating the process until the desired resolution is reached. **LapGAN** [DCF+15] uses a Laplacian pyramid as the iterative building block, by first upsampling the lower dimensional samples using a simple upsampling operation, such as smoothed upsampling, and then using a conditional generator to produce a residual to be added to the upsampled version to produce the higher resolution sample. In turn, this higher resolution sample can then be provided to another LapGAN layer to produce another, even higher resolution

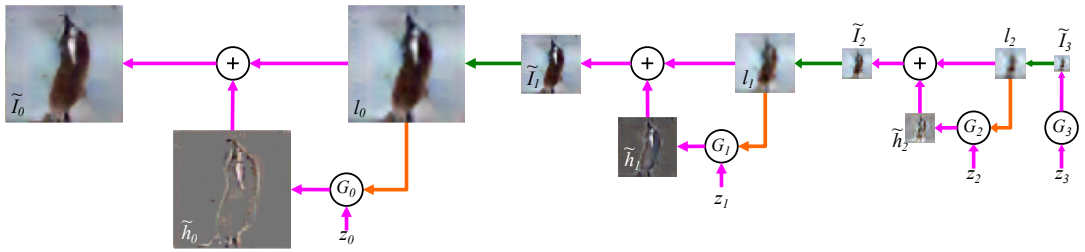


Figure 26.13: LapGAN generation algorithm: the generation process starts with a low dimension sample, which gets upscaled and residually added to the output of a generator at a higher resolution. The process gets repeated multiple times. From Figure 1 of [DCF+15]. Used with kind permission of Emily Denton.

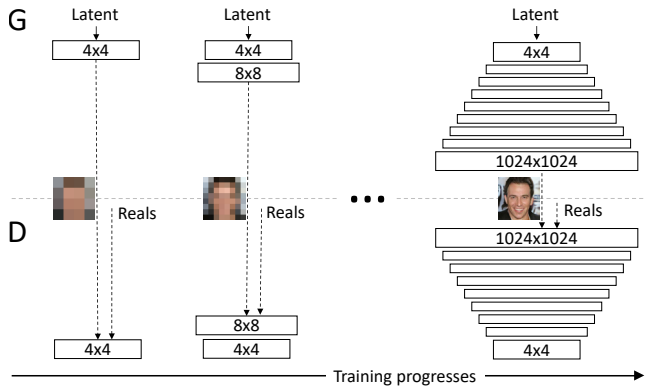


Figure 26.14: ProgressiveGAN training algorithm. The input to the discriminator at the bottom of the figure is either a generated image, or a real image (denotes as 'Reals' in the figure) at the corresponding resolution. From Figure 1 of [Kar+18]. Used with kind permission of Tero Karras.

sample, and so on — this process is shown in Figure 26.13. In LapGAN, a different generator and critic are trained for each iterative block of the model; in ProgressiveGAN [Kar+18] the lower resolution generator and critic are "grown", by becoming part of the generator and critic used to learn to generate higher resolution samples. The higher resolution generator is obtained by adding new layers on top of the last layer of the lower resolution generator. A residual connection between an upscaled version of the lower dimensional sample and the output of the newly created higher resolution generator is added, which is annealed from 0 to 1 in training — transitioning from using the upscaled version of the lower dimensional sample early in training, to only using the sample of the higher resolution generator at the end of training. A similar change is done to the discriminator. Figure 26.14 shows the growing generator and discriminators in ProgressiveGAN training.

26.6.5 Regularization

Regularizing both the discriminator and the generator has by now a long tradition in GAN training. Regularizing GANs can be justified from multiple perspectives: theoretically, as it has been shown to

be tied to convergence analysis [MGN18b]; empirically, as it has been shown to help performance and stability in practice [RMC15; Miy+18c; Zha+19c; BDS18]; and intuitively, as it can be used to avoid overfitting in the discriminator and generator. Regularization approaches include adding noise to the discriminator input [AB17], adding noise to the discriminator and generator hidden features [ZML16], using BatchNorm for the two players [RMC15], adding dropout in the discriminator [RMC15], spectral normalization [Miy+18c; Zha+19c; BDS18], and gradient penalties (penalizing the norm of the discriminator gradient with respect to its input $\|\nabla_{\boldsymbol{x}} D_{\boldsymbol{\phi}}(\boldsymbol{x})\|^2$) [Arb+18; Fed+18; ACB17; Gul+17]. Often regularization methods help training regardless of the type of loss function used, and have been shown to have effects both on training performance as well as the stability of the GAN game. However, improving stability and improving performance in GAN training can be at odds with each other, since too much regularization can make the models very stable but reduce performance [BDS18].

26.6.6 Scaling up GAN models

By combining many of the architectural tricks discussed thus far — very large residual networks, self attention, spectral normalization both in the discriminator and the generator, BatchNormalization in the generator — one can train GANs to generating diverse, high quality data, as done with BigGAN [BDS18], StyleGAN [Kar+20c], and alias-free GAN [Kar+21]. Beyond combining carefully chosen architectures and regularization, creating large scale GANs also require changes in optimization, with large batch sizes being a key component. This furthers the view that the key components of the GAN game — the losses, the parameterization of the models, and the optimization method — have to be viewed collectively rather than in isolation.

26.7 Applications

The ability to generate new plausible data enables a wide range of applications for GANs. This section will look at a set of applications that aim to demonstrate the breadth of GANs across different data modalities, such as images (Section 26.7.1), video (Section 26.7.2), audio (Section 26.7.3), and text (Section 26.7.4), and include applications such as imitation learning (Section 26.7.5), domain adapation (Section 26.7.6), and art (Section 26.7.7).

26.7.1 GANs for image generation

The most widely studied application area is in image generation. We focus on the translation of one image to another using either paired or unpaired datasets. There are many other topics related to image GANs that we do not cover, and a more complete overview can be found in other sources, such as [Goo16] for the theory and [Bro19] for the practice. A JAX notebook which uses a small pretrained GAN to generate some face images can be found at GAN_JAX_CelebA_demo.ipynb. We show the progression of quality in sample generation of faces using GANs in Figure 26.15. There is also increasing need to consider the generation of images with regards to the potential risks they can have when used in other domains, which involve discussions of synthetic media and **deep fakes**, and sources for discussion include [Bru+18; Wit].

Figure 26.15: Increasingly realistic synthetic faces generated by different kinds of GAN, specifically (from left to right): original GAN [Goo+14], DCGAN [RMC15], CoupledGAN [LT16], ProgressiveGAN [Kar+18], StyleGAN [KLA19]. Used with kind permission of Ian Goodfellow. An online demo, which randomly generates face images using StyleGAN, can be found at `https://thispersondoesnotexist.com`.

26.7.1.1 Conditional image generation

Class-conditional image generation using GANs has become a very fruitful endeavor. BigGAN [BDS18] carries out class-conditional generation of ImageNet samples across a variety of categories, from dogs and cats to volcanoes and hamburgers. StyleGAN [KLA19] is able to generate high quality images of faces at high resolution by learning a conditioning style vector and the ProgressiveGAN architecture discussed in Section 26.6.4. By learning the conditioning vector they are able to generate samples which interpolate between the styles of other samples, for example by preserving coarser style elements such as pose or face shape from one sample, and smaller scale style elements such as hair style from another; this provides fine grained control over the style of the generated images.

26.7.1.2 Paired image-to-image generation

We have discussed in Section 26.4 how using paired data of the form $(\boldsymbol{x}_n, \boldsymbol{y}_n)$ can be used to build conditional generative models of $p(\boldsymbol{x}|\boldsymbol{y})$. In some cases, the conditioning variable $\boldsymbol{y}$ has the same size and shape as the output variable $\boldsymbol{x}$. The resulting model $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{y})$ can then be used to perform **image to image translation**, as illustrated in Figure 26.16, where $\boldsymbol{y}$ is drawn from the **source domain**, and $\boldsymbol{x}$ from the **target domain**. Collecting paired data of this form can be expensive, but in some cases, we can acquire it automatically. One such example is image colorization, where a paired dataset can easily be obtained by processing color images into grayscale images (see e.g., [Jas]).

A conditional GAN used for paired image-to-image translation was proposed in [Iso+17], and is known as the **pix2pix** model. It uses a U-net style architecture for the generator, as used for semantic segmentation tasks. However, they replace the batch normalization layers with instance normalization, as in neural style transfer.

For the discriminator, pix2pix uses a **patchGAN** model, that tries to classify local patches as being real or fake (as opposed to classifying the whole image). Since the patches are local, the discriminator is forced to focus on the style of the generated patches, and ensure they match the statistics of the target domain. A patch-level discriminator is also faster to train than a whole-image discriminator, and gives a denser feedback signal. This can produce results similar to Figure 26.16

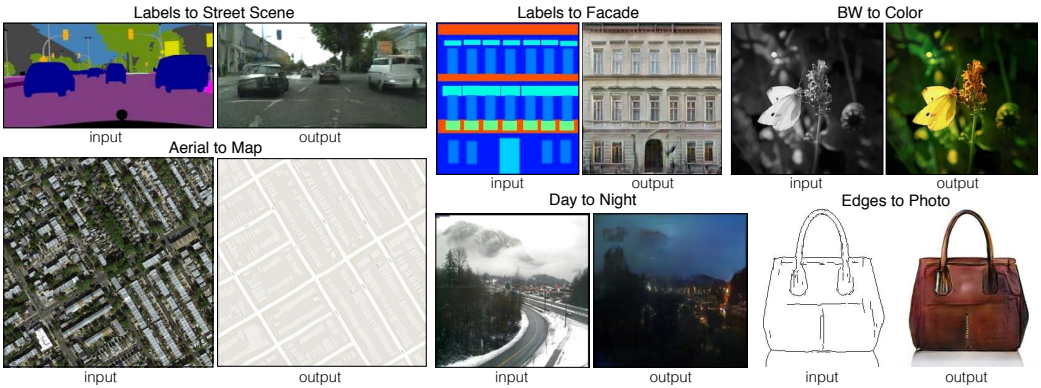


Figure 26.16: Example results on several image-to-image translation problems as generated by the pix2pix conditional GAN. From Figure 1 of [Iso+17]. Used with kind permission of Phillip Isola.

(depending on the dataset).

26.7.1.3 Unpaired image-to-image generation

A major drawback of conditional GANs is the need to collect paired data. It is often much easier to collect **unpaired data** of the form $\mathcal{D}_x = \{\boldsymbol{x}_n : n = 1 : N_x\}$ and $\mathcal{D}_y = \{\boldsymbol{y}_n : n = 1 : N_y\}$. For example, $\mathcal{D}_x$ might be a set of daytime images, and $\mathcal{D}_y$ a set of night-time images; it would be impossible to collect a paired dataset in which exactly the same scene is recorded during the day and night (except using a computer graphics engine, but then we wouldn't need to learn a generator).

We assume that the datasets $\mathcal{D}_x$ and $\mathcal{D}_y$ come from the marginal distributions $p(\boldsymbol{x})$ and $p(\boldsymbol{y})$ respectively. We would then like to fit a joint model of the form $p(\boldsymbol{x}, \boldsymbol{y})$, so that we can compute conditionals $p(\boldsymbol{x}|\boldsymbol{y})$ and $p(\boldsymbol{y}|\boldsymbol{x})$ and thus translate from one domain to another. This is called **unsupervised domain translation**.

In general, this is an ill-posed problem, since there are an infinite number of different joint distributions that are consistent with a set of marginals $p(\boldsymbol{x})$ and $p(\boldsymbol{y})$. We can try, however, to learn a joint distribution such that samples from it satisfy additional constraints. For example, if G is a conditional generator that maps a sample from $\mathcal{X}$ to $\mathcal{Y}$, and F maps a sample from $\mathcal{Y}$ to $\mathcal{X}$, it is reasonable to require that these be inverses of each other, i.e., $F(G(\boldsymbol{x})) = \boldsymbol{x}$ and $G(F(\boldsymbol{y})) = \boldsymbol{y}$. This is called a **cycle consistency** loss [Zhu+17]. We can encourage G and F to satisfy this constraint by using a penalty term on the difference between the starting image and the image we get after going through this cycle:

$$\mathcal{L}_{\text{cycle}} = \mathbb{E}_{p(\boldsymbol{x})}||F(G(\boldsymbol{x})) - \boldsymbol{x}||_1 + \mathbb{E}_{p(\boldsymbol{y})}||G(F(\boldsymbol{y})) - \boldsymbol{y}||_1 \tag{26.64}$$

To ensure that the outputs of G are samples from $p(\boldsymbol{y})$ and those of F are samples from $p(\boldsymbol{x})$, we use a standard GAN approach, introducing discriminators D_X and D_Y, which can be done using any choice of GAN loss $\mathcal{L}_{\text{GAN}}$, as visualized in Figure 26.17. Finally, we can optionally check that applying the conditional generator to images from its own domain does not change them:

$$\mathcal{L}_{\text{identity}} = \mathbb{E}_{p(\boldsymbol{x})}\|\boldsymbol{x} - F(\boldsymbol{x})\|_1 + \mathbb{E}_{p(\boldsymbol{y})}\|\boldsymbol{y} - G(\boldsymbol{y})\|_1 \tag{26.65}$$

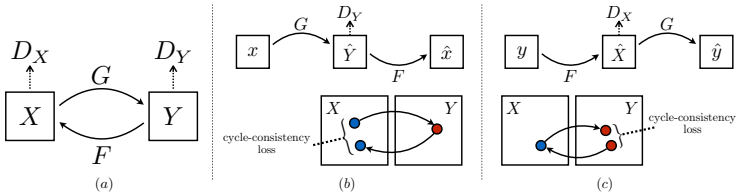


Figure 26.17: Illustration of the CycleGAN training scheme. (a) Illustration of the 4 functions that are trained. (b) Forwards cycle consistency from $\mathcal{X}$ back to $\mathcal{X}$. (c) Backwards cycle consistency from $\mathcal{Y}$ back to $\mathcal{Y}$. From Figure 3 of [Zhu+17]. Used with kind permission of Jun-Yan Zhu.

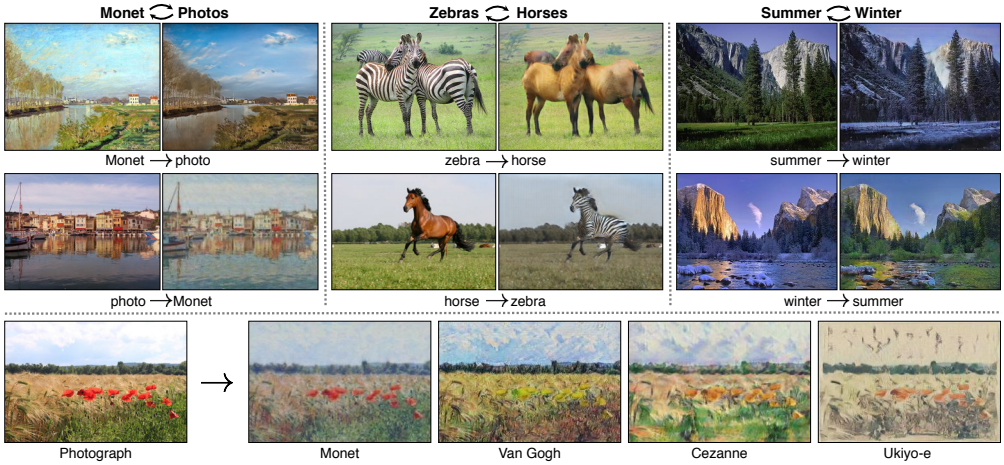


Figure 26.18: Some examples of unpaired image-to-image translation generated by the CycleGAN model. From Figure 1 of [Zhu+17]. Used with kind permission of Jun-Yan Zhu.

We can combine all three of these consistency losses to train the translation mappings F and G, using hyperparameters λ_1 and λ_2:

$$\mathcal{L} = \mathcal{L}_{\text{GAN}} + \lambda_1 \mathcal{L}_{\text{cycle}} + \lambda_2 \mathcal{L}_{\text{identity}} \tag{26.66}$$

CycleGAN results on various datasets are shown in Figure 26.18. The bottom row shows how CycleGAN can be used for **style transfer**.

26.7.2 Video generation

The GAN framework can be expanded from individual images (frames) to videos; the techniques used to generate realistic images can also be applied to generate videos, with additional techniques required to ensure *spatio-temporal consistency*. Spatio-temporal consistency is obtained by ensuring that the discriminator has access to the real data and generated sequences in order, thus penalizing the generator when generating realistic individual frames without respecting temporal order [SMS17;

Sai+20; CDS19; Tul+18]. Another discriminator can be employed to additionally ensure each frame is realistic [Tul+18; CDS19]. The generator itself needs to have a temporal element, which is often implemented through a recurrent component. As with images, the generation framework can be expanded to video-to-video translation [Ban+18; Wan+18], encompassing applications such as motion transfer [Cha+19a].

26.7.3 Audio generation

Generative models have been demonstrated in the tasks of generating audio waveforms, as well as for the task of text-to-speech (TTS) generation. Other types of generative models, such as autoregressive models, such as WaveNet [oor+16] and WaveRNN [Kal+18b] have been developed for these applications, although autoregressive models are difficult to parallelize over time since they predict each time step of the audio sequentially and can be computationally expensive and too slow to be used in practice. GANs provide an alternative approach for these tasks and other paths for addressing these concerns.

Many different GAN architectures have been developed for audio-only generation, including generation of single note recordings from instruments by GANSynth, a vocoder model that uses GANs to generate magnitude spectrograms from mel-spectrograms [Eng+18], in voice conversion using a modified CycleGAN discussed above [Kan+20], and the direct generation of raw audio in WaveGAN [DMP18].

Initial work on GANs for TTS was developed [Yan+17] whose approach is similar to conditional GANs for image generation (see Section 26.7.1.2), but uses 1d convolution instead of 2d. More recent GANs such as GAN-TTS [Bińkowski+19] use more advanced architectures and discriminators that operate at multiple frequency scales that have performance that now matches the best performing autoregressive models when assessed using mean opinion scores. In both the direct-audio generation, the ability of GANs to allow faster generation and different types of context is the advantage that makes them advantageous compared to other models.

26.7.4 Text generation

Similar to image and audio domains, there are several tasks for text data for which GAN-based approaches have been developed, including conditional text generation and text-style transfer. Text data are often represented as discrete values, at either the character level or the word-level, indicating membership within a set of a particular vocabulary size (alphabet size, or number of words). Due to the discrete nature of text, GAN models trained on text are *explicit*, since they explicitly model the probability distribution of the output, rather than modeling the sampling path. This is unlike most GAN models of continuous data such as images that we have discussed in the chapter so far, though explicit GANs of continuous data do exist [Die+19b].

The discrete nature of text is why maximum likelihood is one of the most common methods of learning generative models of text. However, models trained with maximum likelihood are often limited to autoregressive models, while like in the audio case, GANs make it possible to generate text in a non-autoregressive manner, making other tasks possible, such as one-shot feedforward generation [Gul+17].

The difficulty of generating discrete data such as text using GANs can be seen looking at their loss function, such as in Equations (26.19), (26.21) and (26.28). GAN losses contain terms of

the form $\mathbb{E}_{q_\theta(\boldsymbol{x})} f(\boldsymbol{x})$, which we not only need to evaluate, but also backpropagate through, by computing $\nabla_\theta \mathbb{E}_{q_\theta(\boldsymbol{x})} f(\boldsymbol{x})$. In the case of implicit distributions given by latent variable models, we used the reparameterization trick to compute this gradient (Equation 26.49). In the discrete case, the reparameterization trick is not available and we have to look for other ways to estimate the desired gradient. One approach is to use the score function estimator, discussed in Section 6.3.4. However, the score function estimator exhibits high gradient variance, which can destabilize training. One common approach to avoid this issue is to pre-train the language model generator using maximum likelihood, and then to fine-tune with a GAN loss which gets backpropagated into the generator using the score-function estimator, as done by Sequence GAN [Yu+17], MaliGAN [Che+17], and RankGAN [Lin+17a]. While these methods spearheaded the use of GANs for text, they do not address the inherent instabilities of score function estimation and thus have to limit the amount of adversarial fine tuning to a small number of epochs and often use a small learning rate, keeping their performance close to that of the maximum-likelihood solution [SSG18a; Cac+18].

An alternative to maximum likelihood pretraining is to use other approaches to stabilize the score function estimator or to use continuous relaxations for backpropagation. ScratchGAN is a word-level model that uses large batch sizes and discriminator regularization to stabilize score function training (these techniques are the same that we have seen as stabilizers for training image GANs) [Md+19]. [Pre+17b] completely avoid the score function estimator and develop a character level model without pre-training, by using continuous relaxations and curriculum learning. These training approaches can also benefit from other architectural advances, e.g., [NNP19] showed that language GANs can benefit from complex architectures such as relation networks [San+17].

Finally, unsupervised text style transfer, mimicking image style transfer, have been proposed by [She+17; Fu+17] using adversarial classifiers to decode to a different style/language, or like [Pra+18] who trains different encoders, one per style, by combining the encoder of a pre-trained NMT and style classifiers, among other approaches.

26.7.5 Imitation learning

Imitation learning takes advantage of observations of expert demonstrations to learn action policies and reward functions of unknown environments by minimizing some form of discrepancy between the learned and the expert behaviors. There are many approaches available, including behavioral cloning [PPG91] that treats this problem as one of supervised learning, and inverse reinforcement learning [NR00b]. GANs are appealing for imitation learning since they provide a way to avoid the difficulty of designing good discrepancy functions for behaviors, and instead learn these discrepancy functions using a discriminator between trajectories generated by a learned agent and observed demonstrations.

This approach, known as generative adversarial imitation learning (**GAIL**) [HE16a] demonstrates the ability to use GANs for complex behaviors in high-dimensional environments. GAIL jointly learns a generator, which forms a stochastic policy, along with a discriminator that acts as a reward signal. Like we saw in the probabilistic development of GANs in the earlier sections, GAIL can also be generalized to multiple f-divergences, rather than the standard Jensen-Shannon divergence used as the standard loss in GANs. This has led to a family of other GAIL variants that use other f-divergences [Ke+19a; Fin+16; Bro+20c], including f-GAIL that aims to also learn the best f-divergence to use [Zha+20e], as well as new analytical insight into the computation and generalization of such approaches [Che+20b].

26.7.6 Domain adaptation

An important task in machine learning is to correct for shifts in the data distribution over time, minimizing some measure of domain shift, as we discuss in Section 19.5.3. Like with the other applications, GANs are popular as ways of avoiding the choice of distance or degree of shift. Both the supervised and unsupervised approaches for image generation we reviewed earlier looked at pixel-level domain adaptation models that perform distribution alignment in raw pixel space, translating source data to the style of a target domain, as with pix2pix and CycleGAN. Extensions of these approaches for the general problem of domain adaptation seek to do this not only in the observed data space (e.g., with pixels), but also at the feature level. One general approach is domain-adversarial training of neural networks [Gan+16b] or adversarial discriminative domain adaptation (ADDA) [Tze+17]. The CyCADA approach of [Hof+18] extends CycleGAN by enforcing both structural and semantic consistency during adaptation using a cycle-consistency loss and semantics losses based on a particular visual recognition task. There are also many extensions that include class conditional information [Tsa+18; Lon+18] or adaptation when the modes to be matched have different frequencies in the source and target domains [BHC19].

26.7.7 Design, art and creativity

Generative models, particularly of images, have added to approaches in the more general area of algorithmic art. The applications in image and audio generation with transfer can also be considered aspects of artistic image generation. In these cases, the goal of training is not generalization, but to create appealing images across different types of visual aesthetics [Sar18]. One example takes style transfer GANs to create visual experiences, in which objects placed under a video are re-rendered using other visual styles in real time [AFG19]. The generation ability has been used to explore alternative designs and fabrics in fashion [Kat+19], and have now also become part of major drawing software to provide new tools to support designers [Ado]. And beyond images, creative and artistic expression using GANs include areas in music, voice, dance, and typography [AI 19].

Part V

Discovery