Generative Models

Wooyong Park

Lecture from STA4124(Lecturer: Kyungwoo Song)

May 10, 2025

This is a document in progress based on a lecture on generative models by Professor Kyungwoo Song. Although I am sharing this document in public to be receptive of any feedback, I must note that this is not a complete document and **do not want the document to be shared without my permission.** Please abstain from sharing this document without my permission.

Contents

1 Introduction(Wk 1-1)

1.1 Generative Models

A **generative model** is a model that can generate new data points from the same distribution as the training data. Indeed, generative models require the knowledge of the joint distribution $p(x)$, for $x \in \mathcal{X}$. They are often divided into two categories: **deep generative models(DGM)** and **probabilistic graphical models(PGM)**.

- **DGM**: use deep neural networks to learn a complex mapping from the latent space to the data space.
- **PGM**: use probabilistic graphical models to represent the interconnected structure of the latent variables mapped to the data space in a simpler way.

1.2 Data and the Goal

- Data: (X, y) or (X)
- Goal : Learn a model that represents the distribution with some observed samples, capturing a hidden or underlying structure of the data
- Could be used for both **Density Estimation** and **Sample Generation**

1.3 Generative AI Paradox

Generative models seem to acquire generation abilities more effectively than understanding, in contrast to human intelligence where generation is usually harder. That Generative AIs still show basic errors in understanding when it displays superhuman capabilities for generation is difficult to understand.

2 Basics of Graphical Models(Wk 1-2)

2.1 Introduction

There are multiple roles that generative models can play:

- Generation
- Debias
- Outlier Detection[1]
- Classification
- Segmentation
- and more!

[1] By estimating the likelihood from the existing data, the generative model can determine whether a data point would be an outlier or not. It must be noted, neverthelss, that not all generative models estimate the likelihood.

The main goal is to capture the hidden or underlying structure of the data. We can first consider **independence** and **causality** the starting point of understanding underlying patterns.

Consider a set of random variables $\{X_1, \ldots, X_n\}$.

- Which variables are independent?
- Which variables are conditionally independent given others?
- What are the marginal distributions of subsets of variables?

We can to these depndence questions with the joint distribution, $P(X_1, \ldots, X_n)$. Marginal distribution is obtained by summing over the joint distribution, and independence can be answered by checking the factorizations of the marginals.

2.2 Directed Graphical Models(DGM)

A directed graphical model is a model that relies on **directed acyclic graphs(DAGs)**. "Acyclic" means that it does not display or form a cyclic relation. All random variables are represented by a node, as in Figure 2.1. We denote the parent nodes of a node X_i by π_i. For example, in Figure 2.1, $\pi_6 = \{5, 2\}$.

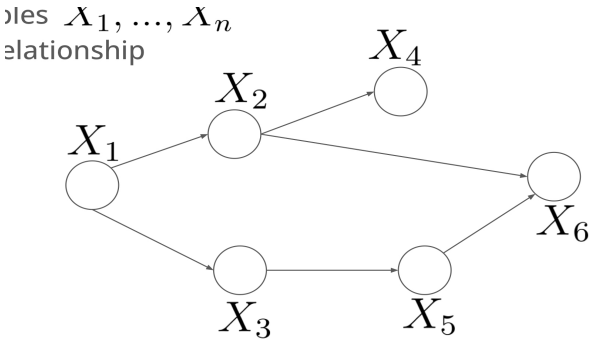


Figure 2.1: Example of DAG

Here, the joint distribution would have the following relation in Equation 2.1.

$$p(x_{1:6}) = p(x_1)p(x_2|x_1)p(X_3|x_1)p(x_4|x_2)p(x_5|x_3)p(x_6|x_2, x_5) \tag{2.1}$$

More generally, the joint distribution given a DAG represents equation 2.2. Note that these kinds of joint probability functions are commonly used in the autogregressive models, which are discussed in section 7. In such case, one should note that directed graphical models come with a particular assumption: only the past data affects the present(i.e., no dependence of the present data on future data). However, such assumption is not always a cost; it allows us to curtail our domain for estimating the model.

$$p(x_{1:n}) = \prod_{i=1}^{n} p(x_i|x_{\pi_i}) \tag{2.2}$$

In detail, consider n binary random variables. Allowing full joint distribution requires us to consider 2^n probabilities. However, graphical models only require us $\sum_{i=1}^{n} 2^{|\pi_i|}$ probabilities.

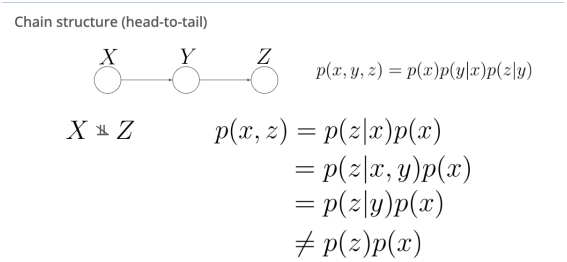
2.3 The Bayesian ball algorithm

For our next step, we want to build a DGM based on the observed independence & conditional independence structure. The Bayesian ball algorithm illustrates the independence properties of each three graphical structures:

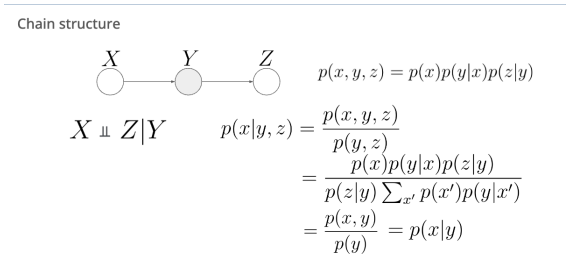
- **Chain Structure(head-to-tail)**
- **Fork Structure(tail-to-tail)**: Common Cause
- **V-Structure(head-to-head, inverse Fork, collider, immoralities)**

2.3.1 Chain Structure

The **chain structure** demonstrates a series of connection where the arrows point to a direction of causality. Note that in DAGs, *the observed variables are filled grey* to represent the conditionality.



(a) Chain structure without observation



(b) Chain structure with observation

Figure 2.2: Chain structure

In figure 2.2, you can see that although X and Z are not independent marginally, but they are conditionally independent given Y. The marginal dependence of X and Z is straightforward, as Z is affected by X through the channel of Y. However, once the variable inside the chain is observed, the other two variables at the opposite ends are independent.

2.3.2 Fork/Common Cause Structure

Also called the **tail-to-tail** structure, the **fork structure** is a structure of two variables(X and Z) having a common cause Y. In this case, X and Z are marginally dependent, but they are conditionally independent given Y, just like the chain structure. This is because Y is the common cause of X and Z, and once we observe Y, the other two variables are independent without any other intermediate factors. In figure 2.3, suppose that Y is the class of the image you are going to generate, and X and Z are the instances - just like a class-instance relation in OOP. - Then, once we observe the class Y, the instances X and Z are independent of each other.

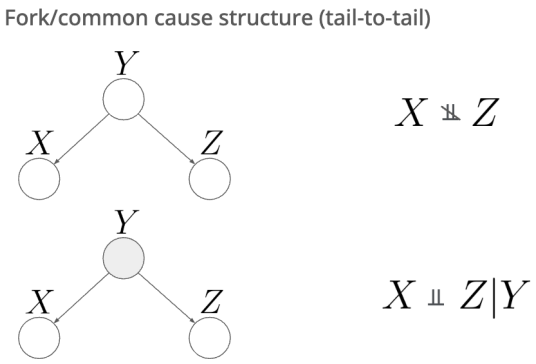


Figure 2.3: Fork structure

2.3.3 V-Structure

The **V-structure** is indeed the most interesting structure among the three. Two variables(X and Z) are the parent nodes of Y, and Y is the child node of X and Z. In v-structure, the parent nodes(X and Z) are conditionally dependent given Y, but they are marginally independent. It is also called the **collider** structure, because Y is the common effect of X and Z, and once Y is observed, there is a selection bias on X and Z's relation.

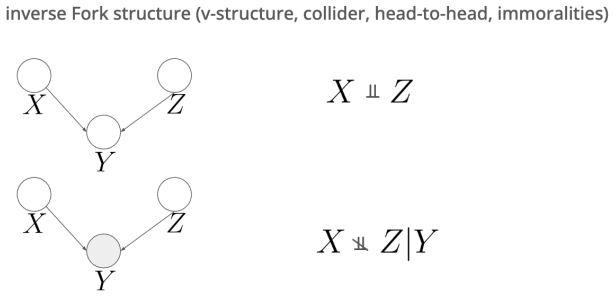


Figure 2.4: V-structure

2.3.4 Caution: Descendants and Ancestry

Note that the descendants of Y in figures 2.2 and 2.4 and the ancestors of Y in figure 2.3 have the same effect as Y.

2.4 D-Separation

D-separation is a criterion for determining whether two sets of nodes are independent given a third set of nodes. Given a directed graph, we can determine whether each pair of nodes is independent or not by checking if there is an open path between the two nodes via the **Bayesian ball algorithm**:

We say that a path is **open** if the following conditions are satisfied:

- If the path is a chain or fork, the middle node is not in the third set of nodes.
- If the path is a v-structure, the middle node is in the third set of nodes.

On the other hand, we say a path is **blocked** if the following conditions are satisfied:

- If the path is a chain or fork, the middle node is in the third set of nodes.
- If the path is a v-structure, the middle node is not in the third set of nodes.

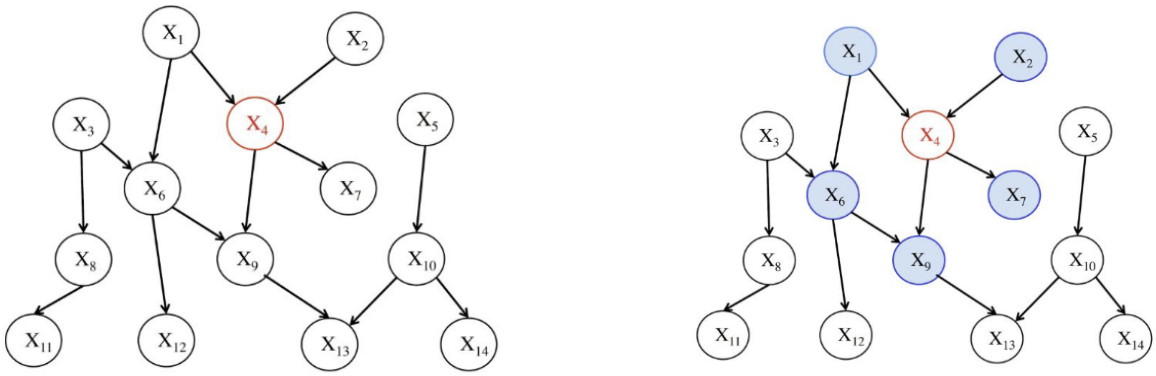
If there is no open path(i.e., all paths are blocked), we say that the two nodes are **d-separated** by the third set of nodes, and thus independent.

2.5 Markov Blanket

Let G denote a directed graph and Π be the set of all nodes in G. Then, the mapping $\mathcal{M} : \Pi \to \mathcal{P}(\Pi)$ is called the **Markov blanket** if

$$p(X_i|X_{\Pi\setminus\mathcal{M}(X_i)}) = p(X_i|X_{\mathcal{M}(X_i)}) \quad \forall i \in \Pi \tag{2.3}$$

where $\mathcal{P}(\Pi)$ is the power set of Π. In other words, a Markov blanket is the set of nodes that makes X_i conditionally independent of all other nodes in the graph. See figure 2.5 for an example of a Markov blanket. With respect to X_4, the Markov blanket is $\{X_1, X_2, X_6, X_7, X_9\}$; condition on those nodes, X_4 is independent of all the other nodes in the graph.



(a) A DGM

(b) The Markov Blanket of X_4

Figure 2.5: Markov Blanket

To determine the Markov blanket of a node is equivalent to identifying the *parents, children, and co-parents(spouse)* of the node. Indeed this is a very useful property of the Markov blanket.

2.6 Plate Notations

Plate notation is a way of representing repetitive structures in probabilistic graphical models more compactly. For example, one parent(class) can have multiple children(instances), and one child can have multiple parents. To avoid repeating the children or parents multiple times, we embrace them with rectangular boxes called **plates**. See figure 2.6 for an example of plate notation.

One application of the plate notation in DGM is to represent the **Gaussian Mixture models(GMM)**. In figure 2.7, the plate on the left represents the mixture of K Gaussians$(\mu_1, \mu_2, \ldots, \mu_K, \Sigma_1, \Sigma_2, \ldots, \Sigma_k)$. On

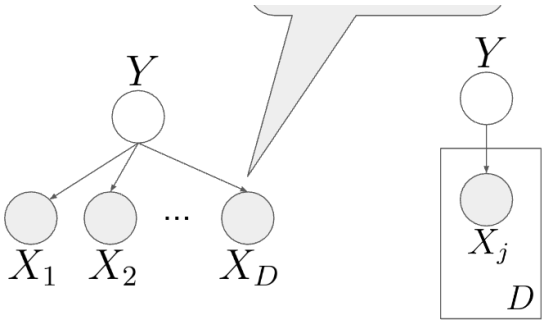


Figure 2.6: Plate Notation

the plate on the right, we can see N samples created from the mixture of K Gaussians. Note that each sample would have its own Gaussian from which it was generated, and thus the origin of x_i is represented by z_i, which follows a categorical distribution with K classes, whose probabilities are represented by $\pi = (\pi_1, \pi_2, \ldots, \pi_K)$.

In other words, the GMM in figure 2.7 follows the following marginal probability distribution:

$$p(x) = sum_{i=1}^{K} \pi_i \mathcal{N}(x|\mu_i, \Sigma_i)$$

where $\mathcal{N}(x|\mu_i, \Sigma_i)$ is the Gaussian distribution with mean μ_i and covariance Σ_i.

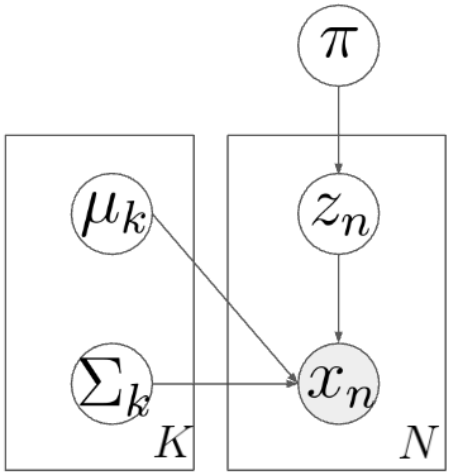


Figure 2.7: Gaussian Mixture Model

2.7 PC Algorithm

The **Peter-Clark(PC) algorithm** is a method for learning the directed acyclic structure of a probabilistic graphical model from data. Due to its nature of **discovering causality**, rather than reling to a prior knowledge of the structure, it is often used in causal inference and requires other set of assumptions(faithfulness, Markov condition, and causal sufficiency) to be satisfied.

Under the assumptions, the PC algorithm takes the following steps:

1. Identify the skeleton: First connect all the nodes in the graph. Then,
   - Check ∃ direct path (1): If two nodes share direct paths, they should be marginally dependent. Thus, if two nodes are marginally independent, erase the direct path between them.
   - Check ∃ direct path (2): If two nodes are marginally dependent but conditionally independent given a third node, they wouldn't share a direct path. If so, erase the direct path between them.
2. Identify immoralities(v-structures): Two nodes are marginally independent but conditionally dependent given a third node iff they have a v-structure. If so, add the direction between these nodes to represent the v-structure.
3. Orient the edges: Once all immoralities are identified, some of the edges(paths) will still remain undirected. Check through all possible paths and if eliminate ones that introduce v-structures or cycles. This should add some information to the graph.

2.7.1 Assumptions(not discussed in the lecture)

- **Faithfulness**: If two variables are conditionally independent in the distribution, there must be a d-separation in the DAG that explains it.
- **Markov Condition**: Given its direct causes (parents), a variable is independent of its non-effects (non-descendants).
- **Causal Sufficiency**: There are no unobserved common causes (confounders).

2.8 Markov Equivalence

Markov equivalence is a property of two directed acyclic graphs that have the same independence relations. That is, two directed acyclic graphs are said to be **Markov equivalent** if they have the same d-separation relations. See figure 2.8 for an example of Markov equivalence.

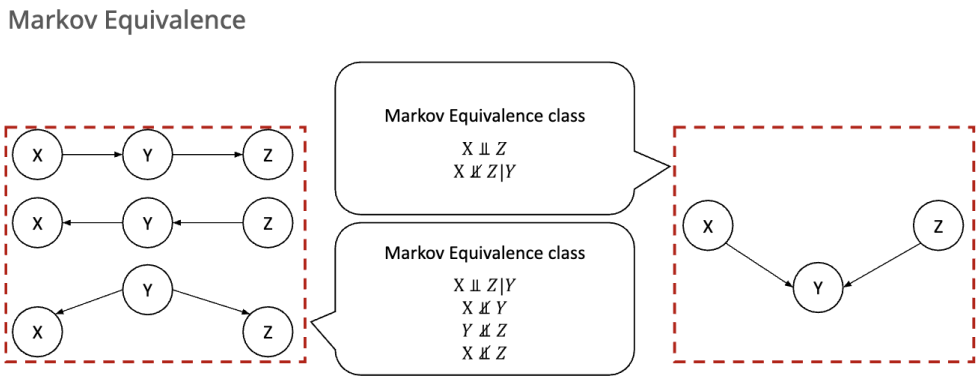


Figure 2.8: Markov Equivalence

3 Variational Inference(Wk 2-1)

One of the goals of generative models is to learn a model that represents the distribution with some observed samples. In Variational Inference (or Variational Bayesian Methods), we are given a set of data $D = \{x_1, \ldots, x_n\}$ and assume that the model $p(x)$ can be represented by a set of parameters, θ, so that $p(x) = p_\theta(x)$.

As in asymptotic statistics, we can construct an MLE, $\hat{\theta} = \text{argmax}_\theta \frac{1}{n} \sum_i^n \log p_\theta(x_i)$.

Furthermore, we also allow latent variables z_i's, which are not observed by the researcher but certainly shapes x_i.

Thus,

$$\begin{aligned} \hat{\theta} &= \text{argmax}_\theta \frac{1}{n} \sum_i^n \log p_\theta(x_i) \\ &= \text{argmax}_\theta \frac{1}{n} \sum_i^n \log \int p_\theta(x_i, z) dz \end{aligned} \tag{3.1}$$

However, the integration in Equation 3.1 is *intractable* because z_i is unobserved.

Thus, we take the following **expected log-likelihood** in Equation 3.2 as an alternative target of maximization.

$$\hat{\theta}_{mele} = \text{argmax}_\theta \frac{1}{n} \sum_i^n E_{z \sim p(z|x_i)} \left[\log p_\theta(x_i, z) \right] = \text{argmax}_\theta \frac{1}{n} \sum_i^n E_{z \sim p(z|x_i)} \left[\log p(x_i|z) + \log p(z) \right] \tag{3.2}$$

In Equation 3.2, $p(z|x_i)$, the distribution of Z given x_i, is called the **posterior**. The posterior seems intractable, too. The essence of VI is to understand (a) that the posterior is much more tractable compared to the joint distribution, $p(x_i, z)$, and (b) that the maximization of the expected log-likelihood is a good alternative to the maximization of the actual log-likelihood.

3.1 Variational Distribution and the Approximation to the Posterior

The log-likelihood of x_i can be expressed as Equation 3.3. Here, $q_i(z)$ is called the **variational distribution**, and the probability distribution $p(z)$ is called the **prior**.

$$\begin{aligned} \log p(x_i) &= \log \int p(x_i|z) p(z) dz \\ &= \log \int p(x_i|z) p(z) \frac{q_i(z)}{q_i(z)} dz \\ &= \log E_{Z \sim q_i(z)} \left[\frac{p(x_i|z) p(z)}{q_i(z)} \right] \end{aligned} \tag{3.3}$$

By Jensen's Inequality, we can obtain the following:

$$
\begin{aligned}
\log p(x_i) &= \log \int p(x_i|z)p(z)dz \\
&= \log E_{Z\sim q_i(z)}\left[\frac{p(x_i|z)p(z)}{q_i(z)}\right] \\
&\geq E_{Z\sim q_i(z)}\left[\log \frac{p(x_i|z)p(z)}{q_i(z)}\right] \\
&= E_{Z\sim q_i(z)}\Big[\log p(x_i|z) + \log p(z)\Big] - E_{Z\sim q_i(z)}\Big[\log q_i(z)\Big]
\end{aligned} \tag{3.4}
$$

Note that in Equation 3.4, the first term is equal to the expected log-likelihood in Equation 3.2, and the latter term $-E_{Z\sim q_i(z)}\Big[\log q_i(z)\Big]$ is the **entropy** of q_i.

For notational convenience, let $H(q_i) \equiv -E_{Z\sim q_i(z)}\Big[\log q_i(z)\Big]$ denote the entropy term. Then,

$$
\log p(x_i) \geq E_{Z\sim q_i(z)}\Big[\log p(x_i|z) + \log p(z)\Big] + H(q_i) \tag{3.5}
$$

The RHS of Equation 3.5 is called **evidence lower bound**[2] or **ELBO**, because it serves as a lower bound to the marginal log-likehlihood. The first thing we can know from Equation 3.5 is that maximizing the expected log-likelihood is a good enough alternative to maximizing $\log p(x_i)$ that we do not know about.

Another interpretation, which is soon discussed in section 3.4, is that maximizing the ELBO also serves as an approximation of $q_i(z)$ to the posterior $p(z|x_i)$.

3.2 Kullback-Leibler Divergence

To begin with, one should know discrepancy measures for probability distributions. Such discrepancy is measured by **divergence**, which is similar to distances. Just like there are multiple ways to measure distance, there are various divergence including *f-divergence*, *H-divergence*, and *IPM(integral probability metrics)*. However, one should note that divergence is not the same as distance - Although distance is always equal to or greater than zero, divergence is not. Moreover, divergence does not always suffice $d(f,g) = d(g,f)$ - In this section, we will discuss **KL divergence**, which is a member of the f-divergence class.

Definition 3.2.1. *The **Kullback-Leibler divergence**[3] of two distribution p and q with a common support $\mathcal{X}$ is defined as*

$$
D_{KL}(p(x)||q(x)) = \begin{cases} \sum_{\mathcal{X}} p(x)\log\frac{p(x)}{q(x)} = -\sum p(x)\log\frac{q(x)}{p(x)} & (x\text{: discrete}) \\ \int_{\mathcal{X}} p(x)\log\frac{p(x)}{q(x)} = -\int p(x)\log\frac{q(x)}{p(x)} & (x\text{: continuous}) \end{cases} \tag{3.6}
$$

The divergence can be interpreted as a **weighted ratio comparison**, in which the log difference between two distributions is weighted by the first distribution. An alternative explanation of the definition can be done by entropy, which is discussed in Appendix C. In general, P is the *target distribution* and Q is the *optimized distribution*. Thus, for a fixed P, the goal is to find Q that minimizes the KL divergence.

[2] In Bayesian statistics, the marginal likelihood of the observable variable is called the **evidence**.

[3] Note that, from the definition, $D_{KL}(p(x)||q(x)) = 0$ if and only if $p = q$.

3.3 Forward KL vs Reversed KL

If the target P is in the first argument of KL divergence, it is called **Forward KL Divergence**; if the target P is in the second argument, it is called the **Reversed KL Divergence**.

$$D_{KL}^{forward} = D_{KL}(P||Q) = \int_{\mathcal{X}} p(x) \log \frac{p(x)}{q(x)} dx \tag{3.7}$$

$$D_{KL}^{reversed} = D_{KL}(Q||P) = -\int_{\mathcal{X}} q(x) \log \frac{p(x)}{q(x)} dx \tag{3.8}$$

In forward KL, optimization is *mean seeking*; howeverm reversed KL is*mode seeking*. In practice, many generative models are based on reversed KL. In the next section, we will discuss how we can use the reversed KL for posterior approximation.

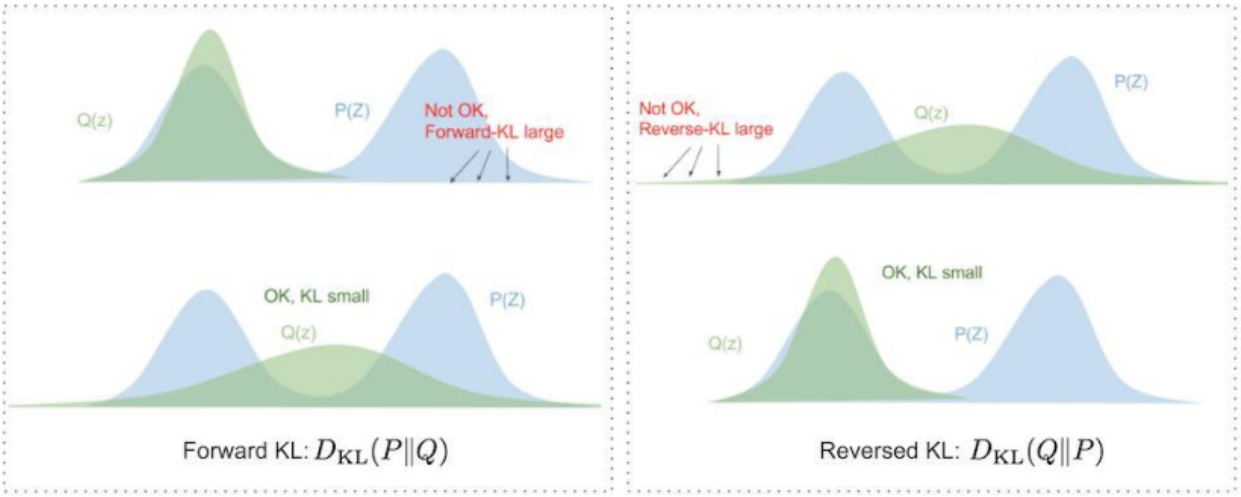


Figure 3.1: Forward and Reversed KL

3.4 Posterior Approximation and Maximizing ELBO

In this section, suppose we want to estimate the posterior $p(z|x_i)$, using $q_i(z)$ and the KL divergence.

Based on reversed KL, one can derive

$$D_{KL}(q_i(z)||p(z|x_i)) = E_{Z\sim q_i(z)}\left[\log \frac{q_i(z)}{p(z|x_i)}\right] \tag{3.9}$$

$$= E_{Z\sim q_i(z)}\left[\log \frac{q_i(z)p(x_i)}{p(z,x_i)}\right] \tag{3.10}$$

$$= -E_{Z\sim q_i(z)}\Big[\log p(x_i|z) + \log p(z)\Big] - H(q_i) + \log p(x_i) \tag{3.11}$$

$$= -L_i(p,q_i) + \log p(x_i) \tag{3.12}$$

where $L_i(p,q_i) \equiv E_{Z\sim q_i(z)}\Big[\log p(x_i|z) + \log p(z)\Big] + H(q_i)$. Note that $L_i(p,q_i)$ is equal to the RHS in Equation 3.5. By moving the first two terms of the right side in Equation 3.11,

$$\log p(x_i) = D_{KL}(q_i(z)||p(z|x_i)) + L_i(p, q_i) \tag{3.13}$$

Since $\log p(x_i)$ is a 'fixed' value, reducing the reversed KL divergence is equivalent to maximizing the sum of expected log-likelihood and the entropy of q. In other words, posterior approximation is equivalent to the maximization of the ELBO, and the problem in Equation 3.14 reduces to that in Equation 3.15.

$$\theta \leftarrow \arg\max_{\theta} \frac{1}{N} \log p_\theta(x_i) \tag{3.14}$$

$$\theta \leftarrow \arg\max_{\theta} \frac{1}{N} L_i(p, q_i) \tag{3.15}$$

The final question is, how do we maximize $L_i(p, q_i)$? First, note that $L_i(p, q_i)$ is the summation of $E_{Z\sim q_i(z)}\Big[\log p(x_i|z) + \log p(z)\Big]$, and $H(q_i)$. Since we are maximizing $L_i(p, q_i)$ with respect to q_i, we are searching for a q_i that has high entropy and is large when $\Big[\log p(x_i|z) + \log p(x)\Big]$ is large. Two main approaches follow this maximization. In the first approach, we assume q_i follows a tractable and convenient distribution, such as the multivariate Gaussian distribution. Then, through gradient ascent on the distribution parameters(e.g. μ_i, σ_i^2 in Gaussian), we obtain the maximizer of ELBO. This is called the **fixed-form variational inference**.

The other approach is the **free-form variational inference**, which usually - but not always - depends on the **mean field assumption** that the posterior factorizes to groups of latent variables:

$$q_{1,\ldots,J}(\mathbf{z}) = \prod_{j=1}^{J} q_j(\mathbf{z}_j) \tag{3.16}$$

This is not to say, however, that we do not assume a particular form of q_j. The allure is that the assumption derives naturally from our *graphical model structure*, which is clearly different from how we impose an arbitrary distribution in fixed-form variational inference. In the next section, we delve into this key strategy.

3.5 Mean Field Variational Inference

If there is a single continuous latent variable that shapes x_i, it might be tractable to model them with Gaussian distributions: $q_1(z) = N(\mu_1, \sigma_1^2), \ldots, q_N(z) = N(\mu_N, \sigma_N^2)$ This is called a **fixed-form variational inference**. On the other hand, there might be layers or groups of latent variables. For example, consider the LDA(Latent Dirichlet Allocations) in figure 3.2.

Such modelling requires simplifying assumptions on the relation of latent variables, so that the optimization becomes tractable. We use the assumption in Equation 3.16. Note that j's denote a group of latent variables; it need not be an individual latent. Also, we are not going to include the subscript i denoting each individual for notational brevity.

Using Equation 3.16, we can rewrite the ELBO as

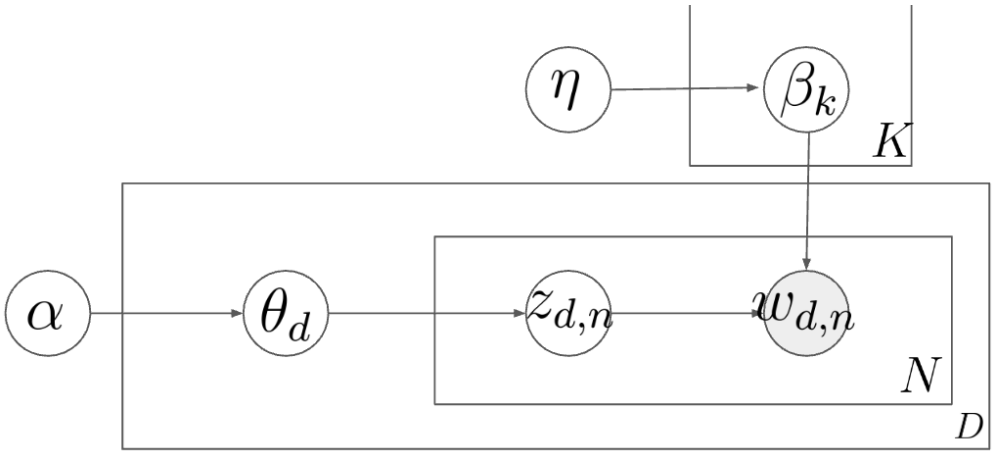


Figure 3.2: Latent Dirichlet Allocations in word clustering

$$ELBO = E_{Z\sim q(\mathbf{z})}\Big[\log p(x|\mathbf{z}) + \log p(\mathbf{z}) - \log q(\mathbf{z})\Big] \tag{3.17}$$

$$= E_{Z\sim \prod_j q_j}\Big[\log p(x|\mathbf{z}) + \log p(\mathbf{z}) - \log \prod_j q_j(z_j)\Big] \tag{3.18}$$

$$= \int_1 \cdots \int_J \Big[\log p(x|\mathbf{z}) + \log p(\mathbf{z})\Big] \prod_j (q_j dz_j) - \int_1 \cdots \int_J \Big[\sum_j \log q_j(z_j)\Big] \prod_j (q_j dz_j) \tag{3.19}$$

As we are maximizing the ELBO with respect to each q_j's, we can rewrite Equation 3.19 as below:

$$ELBO = \int_j q_j \int_{k\neq j} \Big[\log p(x|\mathbf{z}) + \log p(\mathbf{z})\Big] \prod_{k\neq j} (q_k dz_k) dz_j - \int_1 \cdots \int_J \Big[\sum_j \log q_j(z_j)\Big] \prod_j q_j (q_j dz_j) \tag{3.20}$$

$$= \int_j q_j \Big(\int_{k\neq j} \Big[\log p(x, \mathbf{z})\Big] \prod_{k\neq j} (q_k dz_k)\Big) dz_j - \int_1 \cdots \int_J \Big[\log q_j(z_j)\Big] \prod_j (q_j dz_j) + C \tag{3.21}$$

$$= \int_j q_j E_{q_{k\neq j}} \Big[\log(p(x, \mathbf{z}))\Big] dz_j - \int_j \log\Big(q_j(z_j)\Big) q_j dz_j + C \tag{3.22}$$

where $C = \int_1 \cdots \int_J \Big[\sum_{k\neq j} \log q_k(z_k)\Big] \prod_k q_k\big(q_k dz_k\big)$ can be considered a constant wrt q_j.[4] Let us focus on the first term of Equation 3.22, $\int_j q_j E_{q_{k\neq j}} \Big[\log(p(x, \mathbf{z}))\Big] dz_j$. The expectation inside would be a function of z_j; we can write

[4] I must admit that the notation of k and j can cause a lot of muddle. Please try to discern the placeholders from the actual q_j's that matter.

$$E_{q_{k\neq j}}\Big[\log(p(x,\mathbf{z}))\Big] = \int_{k\neq j} \log(p(x,\mathbf{z})) \prod_{k\neq j}\big(q_k dz_k\big) \tag{3.23}$$

$$= \int_{k\neq j} \log\big(p(x,z_j)p(z_{k\neq j}|x,z_j)\big) \prod_{k\neq j}\big(q_k dz_k\big) \tag{3.24}$$

$$= \int_{k\neq j} \log p(x,z_j) \prod_{k\neq j}\big(q_k dz_k\big) + \int_{k\neq j} \log p(z_{k\neq j}|x,z_j) \prod_{k\neq j}\big(q_k dz_k\big) \tag{3.25}$$

$$= \log p(x,z_j) + C' \tag{3.26}$$

$$\triangleq \log \tilde{p}(x,z_j) \tag{3.27}$$

Note that $C' = \int_{k\neq j} \log p(z_{k\neq j}|x,z_j) \prod_{k\neq j}\big(q_k dz_k\big)$ can be considered a constant because it does not depend on our choice of q_j.[5]

Then, Equation 3.22 can be rewritten as

$$ELBO = \int_j q_j \log \tilde{p}(x,z_j) dz_j - \int_j \log\Big(q_j(z_j)\Big) q_j dz_j + C \tag{3.28}$$

$$= \int_j q_j \log \frac{\tilde{p}(x,z_j)}{q_j(z_j)} dz_j + C \tag{3.29}$$

$$= -D_{KL}(q_j||\tilde{p}(x,z_j)) + C \tag{3.30}$$

Then, setting $q_j \propto \tilde{p}(x,z_j)$ - i.e., a **variational distribution** that is proportional to $\tilde{p}$ - will minimize the KL divergence and maximize the ELBO. From Equation 3.27, the minimization can be rewritten as follows:

$$\begin{aligned} q_j &\propto \tilde{p}(x,z_j) \\ \log q_j &\propto \log \tilde{p}(x,z_j) = E_{q_{k\neq j}}\big[\log(p(x,\mathbf{z}))\big] \\ q_j &\propto \exp(E_{q_{k\neq j}}\big[\log(p(x,\mathbf{z}))\big]) \\ q_j^* &\triangleq \frac{\exp(E_{q_{k\neq j}}\big[\log(p(x,\mathbf{z}))\big])}{\int_j \exp(E_{q_{k\neq j}}\big[\log(p(x,\mathbf{z}))\big])dz_j} \end{aligned} \tag{3.31}$$

where q_j^* is the minimization solution to the KL divergence.

In Equation 3.31, calculating the numerator is not difficult at all, given the model structure[6] However, the denominator, $\int_j \exp(E_{q_{k\neq j}}\big[\log(p(x,\mathbf{z}))\big])dz_j$, is the crux. Since we have no analytical solution, we rely on sampling methods, which are discussed in section 4.

3.6 EM Algorithm

The **Expectation-Maximization(EM) algorithm** is a method for finding maximum likelihood estimates of parameters in probabilistic models with latent variables. It can be viewed as a special case of variational inference, where the variational distribution is chosen to be a point mass at the maximum likelihood estimate

[5] Indeed, it is a function of z_j since $p(z_{k\neq j}|x,z_j)$ is. However, this does not affect our optimization over q_j.

[6] For example, if we assume a Bayesian Gaussian Mixture Model delineated in section 3.7, the model depiction in figure 3.4 readily provides us with the joint probability function.

of the latent variables. But first, let us understand the EM algorithm in a more intuitive way before we compare it with variational inference.

The EM algorithm consists of two steps: the E-step and the M-step. In the E-step, we compute the expected value of the complete data log-likelihood given the observed data and the current parameter estimates. In the M-step, we maximize this expected value with respect to the parameters.

3.6.1 Basic Idea

Let $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ be the observed data and $\mathbf{z} = (z_1, z_2, \ldots, z_n)$ be the latent or missing variables.[7]

Then, we have the identity

$$p(\mathbf{z}|\theta, \mathbf{x}) = \frac{p(\mathbf{x}, \mathbf{z}|\theta)}{p(\mathbf{x}|\theta)} \tag{3.32}$$

We call $p(\mathbf{x}, \mathbf{z}|\theta)$ the **complete likelihood** and $p(\mathbf{x}|\theta)$ the **observed likelihood**.

Our goal is to maximize the observed likelihood $p(\mathbf{x}|\theta)$ with respect to θ using the complete likelihood $p(\mathbf{x}, \mathbf{z}|\theta)$.

We use an **auxiliary function**,

$$Q(\theta|\theta_0, \mathbf{x}) = \mathbb{E}_{\theta_0}[\log p(\mathbf{x}, \mathbf{Z}|\theta)|\theta_0, \mathbf{x}] \tag{3.33}$$

for an arbitrary but fixed θ_0.

Calculating the function Q, which is the expected value of the complete log likelihood conditioned on the observed data and the arbitrary parameters, is the E-step. The M-step is to maximize this Q wrt θ. Repeating these steps until convergence will yield our desired estimate of θ:

Algorithm 3.1 Expectation-Maximization Algorithm

Input: $\mathbf{x}$, θ_0
while not converged **do**
  E-step: $Q(\theta|\theta_t, \mathbf{x}) = \mathbb{E}_{\theta_0}[\log p(\mathbf{x}, \mathbf{Z}|\theta)|\theta_t, \mathbf{x}]$
  M-step: $\theta_{t+1} = \arg\max_\theta Q(\theta|\theta_t, \mathbf{x})$
end while

A well-known fact is that given the regualrity conditions and a *unimodal* likelihood, the EM algorithm converges to the maximum likelihood estimator(MLE). Generally, however, the EM algorithm is not guaranteed to converge to the global maximum of the likelihood function, but it is guaranteed to converge to a local maximum.

3.6.2 Mixture of Bernoulli Distributions

Suppose there are N binary data points $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, where $x_i \in \{0, 1\}$ are drawn from K different Bernoulli distributions with parameters $\mathbf{k} = (k_1, k_2, \ldots, k_K)$. Also, let $\theta = (\theta_1, \theta_2, \ldots, \theta_K)$ be the probability of choosing each Bernoulli distribution which would sum to 1. Then

[7] Note that the EM algorithm can be used for cases where some subset of a random sample is missing.

$$p(x) = \prod_{i=1}^{N} \sum_{k=1}^{K} \theta_k p(x_i|k) \tag{3.34}$$

$$\log p(x) = \sum_{i=1}^{N} \log \left(\sum_{k=1}^{K} \theta_k p(x_i|k) \right) \tag{3.35}$$

TBU

3.7 Bayesian Gaussian Mixture Models

Standard **Gaussian mixture models(GMM)** are shaped by three set of key parameters: (1) probability parameters for multinomial, (2-3) mean and variance parameters for each possible Gaussian distribution. In figure 3.3, which depicts a GMM, π is a $k \times 1$ vector that each element sums up to one. μ, Σ are $k\times$ vector and $k \times k$ matrix, respectively for the Gaussian parameters, mean and covariance. In such case, the target of model learning will be to estimate π, μ and Σ.

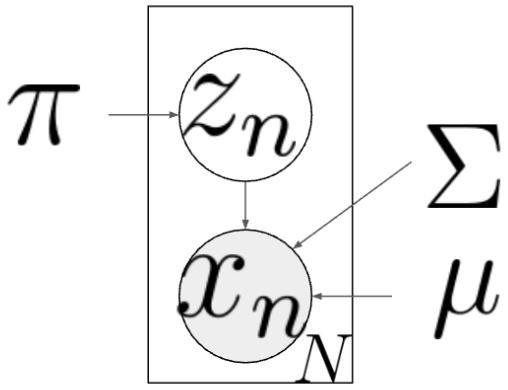


Figure 3.3: Gaussian Mixture Models

On the other hand, **Bayesian Gaussian Mixture Models(BGMM)** has a prior modeling for the proportion, mean, and covariance; which is usually accompanied by Dirichlet and Gaussian Wishart distributions.[8] In the case of figure 3.4, each parameters - π, Λ, and μ - are modeled by the following distributions.

- Dirichlet of order K: $p(\pi) = \text{Dir}(\pi|\alpha_0)$
- k^{th} Gaussian Mean: $p(\mu_k|\Lambda_k) = N(\mu_k|m_0, (\beta_0\Lambda_k)^{-1})$
- k^{th} Gaussian Precision: $p(\Lambda_k) = \mathcal{W}(\Lambda_k|W_0, v_0)$

As usual, we want the posterior, but to obtain it, we first set the variational posterior, $q(z, \pi, \mu, \Lambda)$. By the mean field assumption, let

$$q(z, \pi, \mu, \Lambda) = q(z)q(\pi, \mu, \Lambda) = q(z)q(\pi)q(\mu, \Lambda) \tag{3.36}$$

[8] See Appendix B

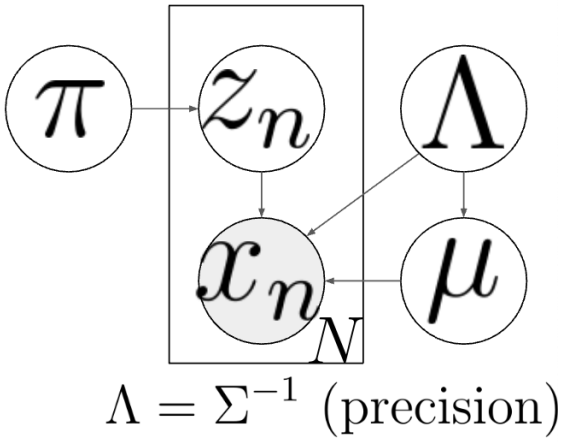


Figure 3.4: Bayesian Gaussian Mixture Models

3.7.1 Finding $q^*(z)$

Based on Equation 3.31, the best variatonal posterior for z will be proportional to $\exp\Big(E_{q_{\pi,\mu,\Lambda}}\big[\log p(x,z,\pi,\mu,\Lambda)\big]\Big)$. In other words,

$$\log q^*(z) \propto E_{q_{\pi,\mu,\Lambda}}\big[\log p(x,z,\pi,\mu,\Lambda)\big] \tag{3.37}$$

The joint likelihood inside the expectation can be rewritten as Equations 3.38:

$$
\begin{aligned}
\log p(x, z, \pi, \mu, \Lambda) &= \log\left[\prod_{k=1}^{K}\prod_{n=1}^{N} p(x_n, z_{nk}, \pi_k, \mu_k, \Lambda_k)\right] \\
&= \log\left[\prod_{k=1}^{K}\prod_{n=1}^{N} p(x_n|z_{nk}, \mu_k, \Lambda_k)p(z_{nk}, \pi_k, \mu_k, \Lambda_k)\right] \\
&= \log\left[\prod_{k=1}^{K}\prod_{n=1}^{N} p(x_n|z_{nk}, \mu_k, \Lambda_k)p(z_{nk}, \pi_k)p(\mu_k, \Lambda_k)\right] \\
&= \log\left[\prod_{k=1}^{K}\prod_{n=1}^{N} p(x_n|z_{nk}, \mu_k, \Lambda_k)p(z_{nk}|\pi_k)p(\pi_k)p(\mu_k, \Lambda_k)\right] \\
&= \sum_{k=1}^{K}\log\left[p(\pi_k)p(\mu_k, \Lambda_k)\right] + \log\left[\prod_{k=1}^{K}\prod_{n=1}^{N} p(x_n|z_{nk}, \mu_k, \Lambda_k)p(z_{nk}|\pi_k)\right] \\
&= \sum_{k=1}^{K}\log\left[p(\pi_k)p(\mu_k, \Lambda_k)\right] + \sum_{k=1}^{K}\log\left[\prod_{n=1}^{N} p(x_n|z_{nk}, \mu_k, \Lambda_k)p(z_{nk}|\pi_k)\right] \\
&= \sum_{k=1}^{K}\log\left[p(\pi_k)p(\mu_k, \Lambda_k)\right] + \sum_{k=1}^{K}\log\left[\prod_{n=1}^{N} p(x_n|z_{nk}, \mu_k, \Lambda_k)p(z_{nk}|\pi_k)\right] \\
&= \sum_{k=1}^{K}\left[\log\left[p(\pi_k)\right] + \log\left[p(\mu_k|\Lambda_k)\right] + \log\left[p(\Lambda_k)\right] + \log\left[\prod_{n=1}^{N} p(x_n|z_{nk}, \mu_k, \Lambda_k)p(z_{nk}|\pi_k)\right]\right] \\
&= \sum_{k=1}^{K}\Bigg[\log\prod_{n=1}^{N} N(x_n|\mu_k, \Lambda_k^{-1})^{z_{nk}} + \log\prod_{n}\pi_k^{z_{nk}} + \log \text{Dirichlet}(\pi|\alpha_0) \\
&\quad + \log N(\mu_k|m_0, (\beta_0\Lambda_k)^{-1}) + \log \mathcal{W}(\Lambda_k|W_0, v_0)\Bigg]
\end{aligned}
\tag{3.38}
$$

In Equations 3.38, only the term written in brown is relevant to the variational posterior of z. Any other terms are constant with respect to z and can be ignored.[9] After some algebra, we can rewrite the term as follows:

$$
q^*(z) \propto \prod_{k=1}^{K}\prod_{n=1}^{N} \rho_{nk}^{z_{nk}} \tag{3.39}
$$

$$
q^*(z) = \prod_{k=1}^{K}\prod_{n=1}^{N} r_{nk}^{z_{nk}} \tag{3.40}
$$

where $\log \rho_{nk} = E_{\pi,\mu,\Lambda}\left[-\frac{1}{2}\log\det(\Lambda_k) - \frac{1}{2}(x_n - \mu_k)^T\Lambda_k(x_n - \mu_k) + \log\pi_k\right]$ and $r_{nk}^{z_{nk}} = \frac{\rho_{nk}^{z_{nk}}}{\sum_{j=1}^{K}\rho_{nj}^{z_{nj}}}$.

3.7.2 Finding $q^*(\pi)$

Based on Equation 3.31, the best variational posterior for π will be proportional to $\exp\left(E_{q_{z,\mu,\Lambda}}\left[\log p(x, z, \pi, \mu, \Lambda)\right]\right)$. In other words,

[9] See my conversation with Claude AI in Appendix D for detailed explanation.

$$\log q^*(\pi) \propto \left(E_{q_{z,\mu,\Lambda}} [\log p(x, z, \pi, \mu, \Lambda)] \right) \tag{3.41}$$

where the joint likelihood inside the expectation can be rewritten as Equations 3.42:

$$\begin{aligned}
\log p(x, z, \pi, \mu, \Lambda) &= \log \left[\prod_{k=1}^{K} \prod_{n=1}^{N} p(x_n, z_{nk}, \pi_k, \mu_k, \Lambda_k) \right] \\
&= \log \left[\prod_{k=1}^{K} \prod_{n=1}^{N} p(x_n | z_{nk}, \mu_k, \Lambda_k) p(\mu_k, \Lambda_k) p(z_{nk} | \pi_k) p(\pi_k) \right] \\
&= \sum_{k=1}^{K} \sum_{n=1}^{N} \left[\log p(x_n | z_{nk}, \mu_k, \Lambda_k) + \log p(\mu_k, \Lambda_k) + \log p(z_{nk} | \pi_k) + \log p(\pi_k) \right] \\
&= \sum_{k=1}^{K} \sum_{n=1}^{N} z_{nk} \left(-\frac{\dim(X)}{2} \log(2\pi) - \frac{1}{2} \log \det(\Lambda_k) - \frac{1}{2} (x_n - \mu_k)^T \Lambda_k (x_n - \mu_k) \right) \\
&+ \sum_{k=1}^{K} \sum_{n=1}^{N} \left(\frac{1}{2} \log \det(\beta_0 \Lambda_k) - \frac{1}{2} (\mu_k - m_0)^T (\beta_0 \Lambda_k)(\mu_k - m_0) \right)_{\text{(excluded irrelevant terms)}} \\
&+ \sum_{k=1}^{K} \sum_{n=1}^{N} \left(z_{nk} \log \pi_k + (\alpha_0 - 1) \log \pi_k \right) \\
&+ \frac{N - \dim(X) - 1}{2} \log \det(\Lambda_k) - \frac{1}{2} tr(W_0^{-1} \Lambda_k)_{\text{(excluded irrelevant terms)}}
\end{aligned} \tag{3.42}$$

In Equations 3.42, only the term written in brown is relevant to the variational posterior of π. Thus,

$$q^*(\pi) \propto E_{z,\mu,\Lambda} \left[\sum_{k=1}^{K} \sum_{n=1}^{N} z_{nk} \log \pi_k + (\alpha_0 - 1) \log \pi_k \right] \tag{3.43}$$

$$q^*(\pi) = \text{Dir}(\pi | \{\alpha_0 + \sum_{n=1}^{N} r_{nk}\}_{k=1}^{K}) \tag{3.44}$$

where r_{nk} is defined similarly to Equation 3.40.

3.7.3 Finding $q^*(\mu, \Lambda)$

For brevity, I leave the results without derivation - try to derive it yourself as an exercise.

$$
\begin{aligned}
q^*(\mu_k, \Lambda_k) \propto E_{z,\pi}\Bigg[\sum_{k=1}^{K} z_{nk}\big(-\frac{1}{2}\log\det(\Lambda_k) - \frac{1}{2}(x_n-\mu_k)^T\Lambda_k(x_n-\mu_k)\big) \\
+ \frac{1}{2}\log\det(\beta_0\Lambda_k) - \frac{1}{2}(\mu_k-m_0)^T(\beta_0\Lambda_k)(\mu_k-m_0) \\
+ \frac{N-\dim(X)-1}{2}\log\det(\Lambda_k) - \frac{1}{2}tr(W_0^{-1}\Lambda)\Bigg]
\end{aligned} \tag{3.45}
$$

$$
q^*(\mu_k, \Lambda_k) = N(\mu_k|m_k, (\beta_k\Lambda_k)^{-1})\mathcal{W}(\Lambda_k|W_k, v_K) \tag{3.46}
$$

4 Markov Chain Monte Carlo(Wk 3-1)

In section 3, we obtained the analytical solution to the approximation of prior, which is demonstrated in Equation 3.31. The crux was the normalizing constant, or the denominator, $\int_j \exp(E_{q_{k\neq j}}[\log p(x, \mathbf{z})])dz_j$. In section 3.7, however, we could circumvent the calculation of denominator since the numerator resembled some known distributions(e.g., Dirichlet, Gaussian-Wishart), - this was due to conjugacy - which per se implies the correct normalizing constant.

However, in many cases, such analytical solution isn't always available and models can be more complex. The goal of **Markov Chain Monte Carlo(MCMC)**, is to directly generate samples from a target distribution - in our case the posterior - and approximate the posterior with the samples. Recall from Bayesian statistics, the posterior is given by the product of the prior and the likelihood, divided by the marginal likelihood:

$$
p(z|x) = \frac{p(x,z)}{p(x)} = \frac{p(x|z)p(z)}{p(x)} \propto p(x|z)p(z) \tag{4.1}
$$

The challenge is that the denominator, $p(x)$, is often intractable to compute. MCMC's goal is to sample from $p(x|z)p(z)$ without explicitly computing the denominator, enabling us to approximate and summarize the posterior distribution. Let's begin with a simple **Monte Carlo approximation** as a warm-up.

4.1 Monte Carlo Approximation and Basic Sampling

Monte Carlo approximation is a technique we use when it is difficult to obtain the analytical solution to an integral or the distribution of a function of some random variable. Given the samples $x_1, \ldots, x_N$ from $p(x)$, we can approximate the distribution of $Y = f(X)$ by the empirical distribution of $f(x_1)\ldots, f(x_n)$. We assume that $p(x)$ is a distribution that is easy to sample from, such as a uniform distribution.[10] The relation between the two random variables and how we can approximate Y's distribution with the empirical distribution of X can be derived as follows:

Theorem 4.1.1. *Suppose the random variable X has a cumulative distribution $S(x)$, and $f(x)$ is a strictly increasing function of X. Then, the cumulative distribution of $Y = f(X)$ is given by $S(f^{-1}(y))$, where $f^{-1}(y)$ is the inverse function of $f(x)$.*[11]

[10] See *4.8. The Method of Monte Carlo* from Hogg et al. (2013)

[11] One can show the assumption that f must be strictly 'increasing' can be relaxed,

Proof.

$$\begin{aligned} P(Y \leq y) &= P(f(X) \leq y) \\ &= P(X \leq f^{-1}(y)) \\ &= S(f^{-1}(y)) \end{aligned}$$

□

One can call the result of Theorem 4.1.1 the **inverse transform sanpling** method.

Example: Let U be a continuous random variable having a standard uniform distribution(i.e., $U \sim \text{Unif}(0,1)$). Then, the random variable $X = F^{-1}(U)$ has a probability distribution characterized by the cumulative distribution function $F(x)$, where $F(x)$ is an invertible function.

$$\begin{aligned} f(x) &= \lambda e^{-\lambda x} \mathbb{I}(x \geq 0) \\ F(X) &= \big(1 - e^{-\lambda x}\big) \mathbb{I}(x \geq 0) \\ F^{-1}(U) &= -\frac{\log(1-U)}{\lambda} \sim Exp(\lambda) \end{aligned}$$

However, the inverse CDF sampling has two major problems:

1. Inverse CDFs are only feasible for a limited number of simple distributions. For example, it is not easy to calculate the inverse CDF for a gneneralized Gamma distribution.
2. Inverse CDF requires the knowledge of *normalizing constant*, such that $\lim_{x \to \infty} F(x) = 1$.

Keeping in mind the limitations of such basic technique of sampling, we could consider **rejection sampling** and **importance sampling**, which are discussed in sections 4.2 and 4.3, respectively.

4.2 Rejection Sampling

Suppose we want to sample from a distribution $p(z)$ - which in our case would be the posterior distribution $p(z|x)$ - and that we can compute $p(z)$ up to some constant, Z_p, such that $p(z) = \frac{1}{Z_p}\tilde{p}(z)$ (i.e., we can compute $\tilde{p}(z)$ for all z).

We set a simpler distribution that we can sample from, $q(z)$, and a constant k such that $kq(z) \geq \tilde{p}(z)$ for all values of z. Here, q is called a **proposal distribution**, or interchangeably, an **instrumental distribution**, and $kq(z)$ is called the **comparison function**.

The algorithm of rejection sampling is as follows:

1. Sample z from $q(z)$.
2. Sample u from $\text{Unif}(0, kq(z))$.
3. If $u < \tilde{p}(z)$, then accept z; otherwise, reject it and go back to step 1.
4. Repeat the above steps until you have enough samples.
5. The samples that are accepted follow the distribution $p(z)$.

The algorithm is illustrated in figure 4.1. The key idea is that we can use the comparison function $kq(z)$ to create a bounding box around the target distribution $\tilde{p}(z)$. Then, we sample uniformly from the bounding box and accept or reject the samples based on whether they fall under the target distribution.

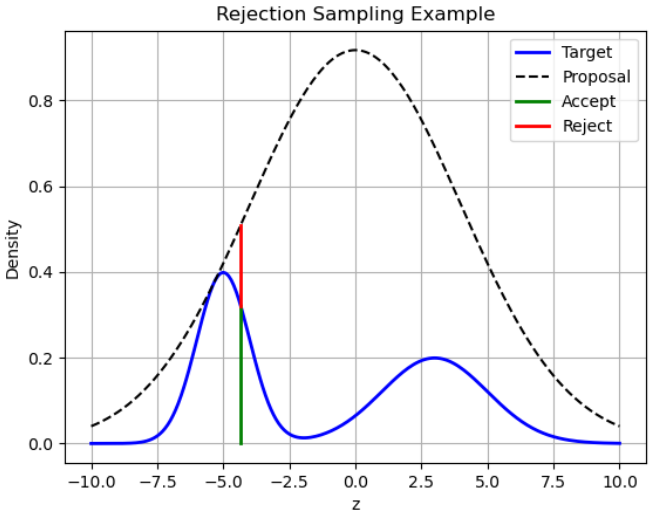


Figure 4.1: Rejection Sampling Algorithm

However, there are serious caveats to rejection sampling: $kq(z)$ should be covering and close to $\tilde{p}(z)$; otherwise the acceptance rate will be low. This means that the size of k matters; the generated samples z are accepted with the probability $\tilde{p}(z)/kq(z)$, and thus the expected probability of being accepted is

$$E[Pr(\text{accept})] = \int \frac{\tilde{p}(z)}{kq(z)} q(z)dz = \frac{1}{k}\int \tilde{p}(z)dz = \frac{Z_p}{k}$$

Although we want a large k that could cover $\tilde{p}(z)$ entirely, we also want k to be small so that the acceptance rate is high. However, as the dimension of Z increases, it is likely that the required k would increase as well.

The other concern is the shape of $q(z)$. Although we have a bell-shaped q in figure 4.1, note that the target distribution is not bell-shaped. This means that certain areas of z are less likely to be accepted than others, which leads to a low acceptance rate and a waste of samples.

4.3 Importance Sampling

Importance sampling is another Monte Carlo method that primarily aims to approximate an integration directly. Recall that expectation of a function is a type of integration with respect to the probability density; thus, we can also approximate an expectation through importance sampling. Consider the expectation below:

$$E[f] = \int f(z)p(z)dz \tag{4.2}$$

where f and p are called the **target function** and the **target distribution**, respectively. Often, the

target distribution is a conditional distribution such as $p(z) = p(z|x)$. - In our case, this can be the posterior. - Since it is difficult to sample from $p(z)$ directly in general, we use a **proposal distribution**, $q(z)$, which must have a support everywhere p does. Note that if $p(z) = p(z|x)$, $q(z)$ will also depend on x.

4.3.1 Direct Importance Sampling

First, consider sampling N number of $f(z)$'s directly from $p(z)$, and then summing them up with $p(z)$ as weights. It is clear that this would be an unbiased estimator for $E_p[f]$. That is,

$$E_p[f] \approx \sum_{i=1}^{N} f(z_i)p(z_i) \tag{4.3}$$

Direct importance sampling assumes that although we cannot sample from $p(z)$ directly, *we can evaluate* $p(z)$ *for any* z. Since we cannot sample from $p(z)$, we use a proposal $q(z)$ instead. The link between $E_p[f]$ and $q(z)$ is given by

$$\begin{aligned} E_p[f] &= \int f(z)p(z)dz && (4.4)\\ &= \int f(z)\frac{p(z)}{q(z)}q(z)dz && (4.5)\\ &\approx \frac{1}{N}\sum_{i=1}^{N} f(z_i)\frac{p(z_i)}{q(z_i)} \triangleq \frac{1}{N}\sum_{i=1}^{N} f(z_i)w_i && (4.6) \end{aligned}$$

Equation 4.6 gives us an insight that we can use $q(z_i)$ to sample $f(z_i)$, and then adjust them by weights($w_i = p(z_i)/q(z_i)$) to calculate the expectation of f. Here, w_i is called the **importance weight** of z_i, which corrects the bias introduced by sampling from a wrong distribution.

The expectation and the variance of this estimator is given by

$$\begin{aligned} E_q\left[\frac{1}{N}\sum_{i=1}^{N} f(z_i)w_i\right] &= \frac{1}{N}\sum_{i=1}^{N}\int f(z_i)w_i q(z_i)dz_i && (4.7)\\ &= \int f(z)wq(z)dz && (4.8)\\ &= \int f(z)p(z)dz = E_p[f] && (4.9) \end{aligned}$$

$$\begin{aligned} \mathbb{V}_q\left[\frac{1}{N}\sum_{i=1}^{N} f(z_i)w_i\right] &= \frac{1}{N}\mathbb{V}_q\Big[f(z)w\Big] && (4.10)\\ &= \frac{1}{N}\Big\{E_q(f^2(z)w^2) - E_q^2(f(z)w)\Big\} && (4.11)\\ &= \frac{1}{N}\left\{\int f^2(z)\frac{p^2(z)}{q^2(z)}q(z)dz - E_p^2[f]\right\} && (4.12) \end{aligned}$$

Based on the equality condition of Cauchy-Schwartz inequality, one can show that

$$\int f^2(z)\frac{p^2(z)}{q^2(z)}q(z)dz = E_p^2[f] \iff q(z) \propto f(z)p(z) \tag{4.13}$$

and thus the variance of the estimator is minimized to zero when $q(z) \propto f(z)p(z)$.

4.3.2 Self-normalized Importance Sampling

In this section, we assume a more prevalent situation where we do not know the normalized distribution $p(z)$, but yet we can evaluate the unnormalized distribution $\tilde{p}(z)$. In this case, we can rewrite Equation 4.6 as follows:

$$E_p[f] \approx \frac{1}{N}\sum_{i=1}^{N} f(z_i)\frac{\tilde{p}(z_i)}{Z_p q(z_i)} \tag{4.14}$$

where $Z_p = \int \tilde{p}(z)dz$

If we use the same approach as in section 4.3.1,

$$\begin{aligned}\int f(z)\frac{\tilde{p}(z)}{q(z)}q(z)dz &= Z_p \int f(z)\frac{p(z)}{q(z)}q(z)dz \\ &\approx Z_p \sum_{i=1}^{N} f(z_i)\frac{p(z_i)}{q(z_i)}\end{aligned}$$

In **self-normalized importance sampling**, we also approximate the denominator Z_p with importance sampling. That is,

$$E_p[f] = \int f(z)p(z)dz = \frac{1}{Z_p}\int f(z)\tilde{p}(z)dz = \frac{\int \left(f(z)\frac{\tilde{p}(z)}{q(z)}\right)q(z)dz}{\int \frac{\tilde{p}(z)}{q(z)}q(z)dz} \tag{4.15}$$

$$\approx \frac{\left(\sum_{i=1}^{N} f(z_i)\tilde{w}_i\right)/N}{\left(\sum_{i=1}^{N}\tilde{w}_i\right)/N} \tag{4.16}$$

where $\tilde{w}_i = \frac{\tilde{p}(z_i)}{q(z_i)}$ is the **unnormalized importance weight**.

We can rewrite equations 4.15 and 4.16 as follows:

$$E_p[f] \approx \sum_{i=1}^{N} f(z_i)W_i \tag{4.17}$$

where $W_i = \frac{\tilde{w}_i}{\sum_{j=1}^{N}\tilde{w}_j}$ is the **normalized importance weight**. Although this estimator is biased:

$$
\begin{aligned}
E_q\left[\sum_{i=1}^{N} f(z_i)W_i\right] &= \int f(z_i)\frac{\tilde{w}_i}{\sum_{j=1}^{N}\tilde{w}_j}q(z_i)dz_i \\
&= \int f(z_i)\frac{\tilde{p}(z_i)/q(z_i)}{\sum_{j=1}^{N}\tilde{p}(z_j)/q(z_j)}q(z_i)dz_i \\
&= \int f(z_i)\frac{\tilde{p}(z_i)}{\sum_{j=1}^{N}\tilde{p}(z_j)/q(z_j)}dz_i \\
&= \int \frac{1}{\sum_{j=1}^{N}\tilde{p}(z_j)/q(z_j)}f(z_i)\tilde{p}(z_i)dz_i \neq \int \frac{1}{Z_p}f(z)\tilde{p}(z)dz
\end{aligned} \tag{4.18}
$$

the bias goes to zero, $\frac{1}{N}\sum_{j=1}^{N}\tilde{p}(z_j)/q(z_j) \to Z_p$, as $N \to \infty$. Thus, the estimator is asymptotically unbiased.

4.4 Markov Chains

4.4.1 Limitations of Rejection Sampling and Importance Sampling

	Rejection Sampling	**Importance Sampling**
Goal	Approximate $p(z)$	Approximate $E_p[f]$
Acceptance Rate	$\tilde{p}(z)/kq(z)$	Every sample is accepted
Optimal $q(z)$	$q(z) = p(z)$	$q(z) \propto f(z)p(z)$
Limitations	Do not work well in high dimensions	

Table 1: Comparison of Rejection Sampling and Importance Sampling

As mentioned in table 1, a common weakness of both sampling methods is that they do not work well in high dimensions. In the case of rejection sampling, the bounding box $kq(z)$ becomes very large as the dimension increases, leading to a low acceptance rate; in the case of importance sampling, the proposal distribution $q(z)$ becomes less representative of the target distribution $p(z)$ as the dimension increases, leading to a high variance in the importance weights.

This is where **Markov Chain Monte Carlo(MCMC)** comes in. MCMC is a class of algorithms that maintains a record of the current state $z^{(\tau)}$ and the proposal distribution $q(z^{(\tau)}|z^{(\tau-1)})$ where the samples $z^{(1)}, z^{(2)}, \ldots, z^{(\tau)}$ form a Markov chain. As opposed to the previous sampling methods, it scales well with the dimension of Z.

4.4.2 Key Concepts of Markov Chains

A (discrete) **Markov chain** is a stochastic process describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event. In other words, the future state of the process is independent of the past states given the present state. A (first-order) **Markov assumption** is defined as follows:

$$
z_{t+1} \perp \mathbf{z}_{1:t-1}|z_t \tag{4.19}
$$

Similarily, a m-order Markov assumption is defined as equation follows:

$$x_{t+1} \perp \mathbf{z}_{1:t-m-1} | \mathbf{z}_{t-m:t} \tag{4.20}$$

We mostly deal with first-order Markov chains, and under equation 4.19, one can write the joint probability of a sequence of events as follows:

$$\begin{aligned} p(\mathbf{z}_{1:T}) &\equiv p(z_1)p(z_2|z_1)p(z_3|\mathbf{z}_{1:2}) \dots p(z_T|\mathbf{z}_{1:T-1}) \\ &\overset{Markov}{=} p(z_1) \prod_{t=2}^{T} p(z_t|z_{t-1}) \end{aligned} \tag{4.21}$$

The probability distribution $p(z_t|z_{t-1})$ is called the **transition probability**, or the transition function, of the Markov chain, and it describes the probability of moving from state x_{t-1} to state x_t. Usually, we denote the transition probability as $T_m(z^{(m)}, z^{(m+1)})$ for $p(z_{m+1}|z_m)$. In the case where transition probabilities are time-invariant, (i.e., $T_m = T_n\big(^{\forall} m \neq n\big)$), we can call the Markov chain a **homogeneous Markov chain**.

4.5 Metropolis-Hastings Algorithm(TBU)

TBU

4.6 Gibbs Sampling(TBU)

TBU

5 Variational Autoencoders(Wk 4-1)

5.1 Autoencoders

An **autoencoder** is a type of an unsupervised neural network that learns to encode the input data into a lower-dimensional representation and then decode it back to the original data. The autoencoder consists of two main components: an **encoder** and a **decoder**. The encoder maps the input data x to a lower-dimensional representation z, while the decoder maps the representation z back to the original data space.

Thus, it is useful for the following tasks:

- Dimensionality reduction
- Learn semantically meaningful representations
- Denoising data
- Image compression
- Reconstruct the input data
- Map high-dimensional data to two dimensions for visualization
- and so on...

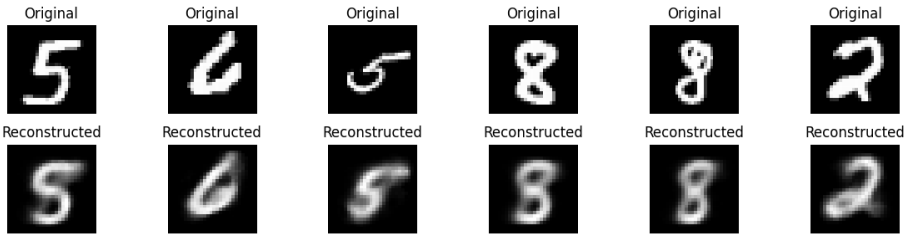


Figure 5.1: An example of MNIST reconstruction with autoencoders

To prevent the autoencoder from copying all the minor details of the input, we usually make the hidden layer smaller than the input layer, resembling a **bottleneck**. The idea, or an assumption, behind the bottleneck structure is the *compactness* of the data structure. Although the data is high-dimensional, we can assume that the underlying factors of variation in the data can be captured by a lower-dimensional latent space. This is in contrast to **flow-based models** and **diffusion models**, which are discussed in section TBU.

In the case where we have one hidden layer only and the activation function is a linear function without the bias term, the autoencoder is equivalent to a **principal component analysis(PCA)**. However, we usually adopt multiple non-linear hidden layers, and the simplist version of that is called a **vanilla autoencoder**. A typical vanilla encoder has fully connected layers with simple activation functions such as ReLU, sigmoid, or tanh, and is optimized using back propagation.

5.1.1 Encoder-Decoder Workflow

- **Encoder**$\big(f_\phi\big)$: compresses the input $\mathbf{x} \in \mathbb{R}^d$ into a latent representation $\mathbf{z} \in \mathbb{R}^k$ where $k < d$.
  $\rightarrow \mathbf{z} = f_\phi(\mathbf{x})$
- **Decoder**$\big(g_\theta\big)$: reconstructs the input $\mathbf{x}$ from the latent representation $\mathbf{z}$.
  $\rightarrow \hat{\mathbf{x}} = g_\theta(\mathbf{z}) = g_\theta(f_\phi(\mathbf{x}))$

Using neural networks for parameterization,

$$f_\phi(\mathbf{x}) = \sigma_\phi(\mathbf{W}_\phi \mathbf{x} + \mathbf{b}_\phi)$$
$$g_\theta(\mathbf{z}) = \sigma_\theta(\mathbf{W}_\theta \mathbf{z} + \mathbf{b}_\theta)$$

where $\mathbf{W}_\phi, \mathbf{W}_\theta, \mathbf{b}_\phi, \mathbf{b}_\theta$ are the weight matrices and bias term for the encoder and decoder, respectively. σ_ϕ and σ_θ are activation functions for the encoder and decoder.

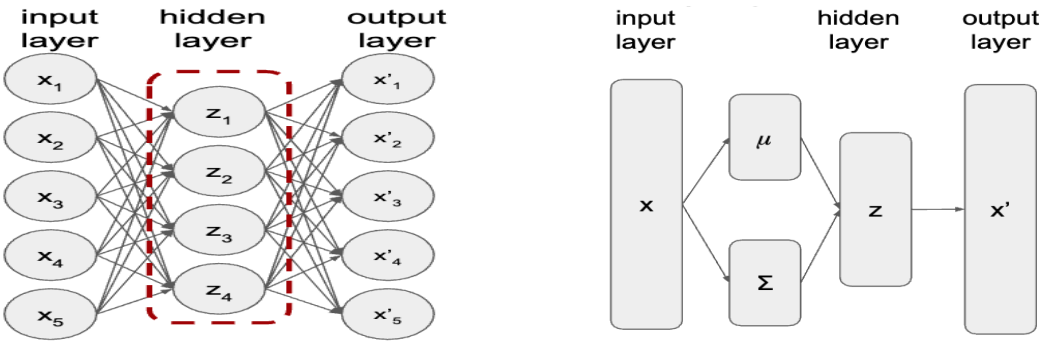
5.1.2 Loss Function(Objective)

Usually, the loss function is defined by the mean squared error of the reconstruction:

$$\min_{\phi,\theta} \mathcal{L}(\phi, \theta) = \min_{\phi,\theta} \frac{1}{N} \sum_{i=1}^{N} \big\| \mathbf{x}_i - g_\theta(f_\phi(\mathbf{x}_i)) \big\|^2$$

5.2 Variational Autoencoders

Variational autoencoders(VAEs) are a type of generative model that combines the principles of variational inference and autoencoders. Note that although AEs are great for compression and reconstruction, they are a *fixed mapping* from the input to output, and thus not ideal for generating new data or modeling uncertainty. Since generation can be thought of as a sampling process, VAEs are designed to learn a probabilistic mapping from the input data to a latent space, and then sample from that latent space to generate new data.



(a) Autoencoders

(b) Variational Autoencoders

5.2.1 Forward Process

In figure 5.2b, we first construct the parameters of a distribution - for example, (μ, Σ) of a Gaussian distribution - from the data. The dimension of these parameters, say d, is usually smaller than the dimension of the data(i.e., dimensionality reduction).

Let the neural network parameters for the encoder(from x to (μ, Σ)) and decoder(from z to x') be denoted as ϕ and θ, respectively. Then, "in principle",we sample $z_1, z_2, \ldots, z_n$ from the normal distribution, $q_\theta(z|x) := \mathcal{N}(\mu(x), \Sigma(x))$, and then pass them through the decoder to generate the data $x'_1, x'_2, \ldots, x'_n$.

5.2.2 Reparameterization Trick

However, the problem is that if we directly sample the latents from $q_\theta(z|x) = \mathcal{N}(\mu(x), \Sigma(x))$, we cannot backpropagate through the sampling process, since $\mathbb{E}_{z\sim q_\phi(z|x_i)}\Big[\log p_\theta(x_i|z)\Big]$ is not differentiable wrt $\mu(x_i)$ and $\Sigma(x_i)$. To solve this problem, we can use the **reparameterization trick**, which allows us to rewrite $\mathbb{E}_{z\sim q_\phi(z|x_i)}\Big[\log p_\theta(x_i|z)\Big]$ as $\mathbb{E}_{e\sim\mathcal{N}(0,1)}\Big[\log p_\theta(x_i|\sigma(x_i)z+\mu(x_i))\Big]$, so that μ and σ are now free from sampling and thus we are able to differentiate wrt them.

The idea is to sample $e_1, e_2, \ldots, e_n$ from a standard normal distribution $\mathcal{N}(0, I)$ and then transform the samples using the parameters of the distribution, so that $z_1 = \mu(x) + \Sigma(x) \odot e_1, \ldots, z_n = \mu(x) + \Sigma(x) \odot e_n$. This way, we can backpropagate through the deterministic function and update the parameters of the encoder and decoder in the backpropagation explained in section 5.2.3. To see a more detailed explanation on what it means that the sampling process is not differentiable, see appendix E.1.

5.2.3 Backward Process

Our goal is to minimize the reconstruction error, $||x_i - x_i'||^2$, but we also want to train the decoder to generate new data from the latent space. To do so, we usually adopt a loss function that is the sum of the reconstruction error and the regularization loss[12]. The loss wrt i-th data instance is defined as follows:

$$L_i = -\mathbb{E}_{z\sim q_\phi(z|x_i)}\Big[\log p_\theta(x_i|z)\Big] + D_{KL}\big[q_\phi(z|x_i)||p(z)\big] \tag{5.1}$$

$$\overset{Repar.}{=} -\mathbb{E}_{e\sim\mathcal{N}(0,I)}\Big[\log p_\theta\big(x_i \mid z = \mu_\phi(x_i) + \sigma_\phi(x_i)\odot e\big)\Big] + D_{\mathrm{KL}}\left[q_\phi(z|x_i)\,||\,p(z)\right] \tag{5.2}$$

where $p(z)$ is the prior distribution of z, which in our case is the standard Gaussian distribution. Notice that through the reparameterization trick, we can now backpropate trough the sampling process, and thus we can update the parameters of the encoder and decoder using the standard backpropagation algorithm.

The minimization of this loss function in equation 5.2 can be interpreted as a trade-off between maximizing $\mathbb{E}_{z\sim q_\phi(z|x_i)}\Big[\log p_\theta(x_i|z)\Big]$ and minimizing the KL divergence, $D_{KL}\big[q_\phi(z|x_i)||p(z)\big]$. In detail,

- $\mathbb{E}_{z\sim q_\phi(z|x_i)}\Big[\log p_\theta(x_i|z)\Big]$: measure of reconstruction
  - This term is the expected log-likelihood of the original data point, x_i, given the latent variable, z, drawn from our encoder's distribution, $q_\phi(z|x_i)$.
  - $p_\theta(x_i|z)$ is usually assumed to be a Gaussian distribution or a Bernoulli distribution, depending on the type of data we are dealing with. *Note that we are assuming simple type of distributions not only for the encoder, but also for the decoder.*
  - It encourages the model to reconstruct the input data well by maximizing the likelihood of the data given the latent representation.

- $D_{KL}\big[q_\phi(z|x_i)||p(z)\big]$: measure of regularization / availability of generation
  - This term is the KL divergence between the encoder's distribution, $q_\phi(z|x_i)$, and the prior distribution, $p(z)$, which is usually a standard normal distribution (or a Bernoulli distribution).
  - It encourages the encoder's distribution to be close to the prior distribution, which helps in generating new data points from the latent space.
  - If Gaussian,

$$D_{KL}\big[q||p\big] = \frac{1}{2}\log\frac{|\Sigma_p|}{|\Sigma_q|} - \frac{1}{2}\mathbb{E}_q\Big[(X-\mu_q)^\top\Sigma_q^{-1}(X-\mu_q)\Big] + \frac{1}{2}\mathbb{E}_q\Big[(X-\mu_p)^\top\Sigma_p^{-1}(X-\mu_p)\Big] \tag{5.3}$$

$$= \frac{1}{2}\Big[\log\frac{|\Sigma_p|}{|\Sigma_q|} - d + (\mu_p-\mu_q)^\top\Sigma_p^{-1}(\mu_p-\mu_q) + tr\{\Sigma_p^{-1}\Sigma_q\}\Big] \tag{5.4}$$

where d is the dimension of z.

[12] Think of regularization loss as the loss from having low generation performance

A few notes should be made here. First, unlike the first term which requires sampling from a distribution, q_ϕ, the second term can be computed directly. For example, if q_ϕ is a Gaussian distribution and p is a standard Gaussian, $D_{KL}\big[q||p\big] = \frac{1}{2}\left[\frac{|\Sigma_p|=1}{|\Sigma_q|} - d + \mu_q^\top \mu_q + tr\{\Sigma_q\}\right]$. For the sampling process related to q_ϕ, we use the reparameterization trick discussed in section 5.2.2. Second, there is a restraint to Σ_p, because the matrix must be symmetric and positive definite. Suppose that Σ_p is a diagonal matrix, then we can either use an exponential function or a softplus matrix to ensure that the diagonal elements are positive:

$$\Sigma_p = \exp\big(W_\Sigma x\big) \text{ or } \Sigma_p = \text{softplus}(W_\Sigma x)$$

where W_Σ is a weight matrix and x is the input data.

To make the covariance matrix positive-definite, we can use the **Cholesky decomposition**, which is a method of decomposing a positive-definite matrix into the product of a lower triangular matrix and its transpose.

$$\Sigma_p = LL^\top \tag{5.5}$$

where L is a lower triangular matrix. Since L should have real and positive diagonal entries, let $L = L' + D$, where L' has zero diagonal terms and D is a diagonal matrix. Then, we can construct D with the same techniques used to construct a diagonal covariance matrix. Figure 5.3 describes the entire process briefly, and the `tensorflow` code for VAE is written in appendix E.2.

5.2.4 Gradient Descent(TBU)

TBU

5.2.5 Relationship to VI

A key feature of VAE is that each data instance, x_i, has its own latent variable z_i, even though we had only a fixed set of encoder parameters, ϕ:

$$X_i \xrightarrow{\phi} \Big(\mu_i, \Sigma_i\Big)$$

Recall that in VI, each instance had its own variational posterior (see, equation 3.31). **Amortized variational inference** circumvents the need to learn a separate variational posterior for each data instance by using a neural network to parameterize the variational posterior. In VAE, this is done in the encoder, which maps the input data to the parameters of the variational posterior distribution. Then, the decoder uses those parameters to sample from the variational posterior and generate new data points. Indeed, one can say that VAEs are a type of amortized VI.

5.3 Conditional VAE

Consider the MNIST dataset in figure 5.1. What we can do with the Vanlilla VAE is to generate new digits by sampling from $Z \sim \mathcal{N}(0, I)$ and then passing the samples through the decoder. If trained well, the images will be similar to the original digits, *but we will have no control over what digits it will produce.*

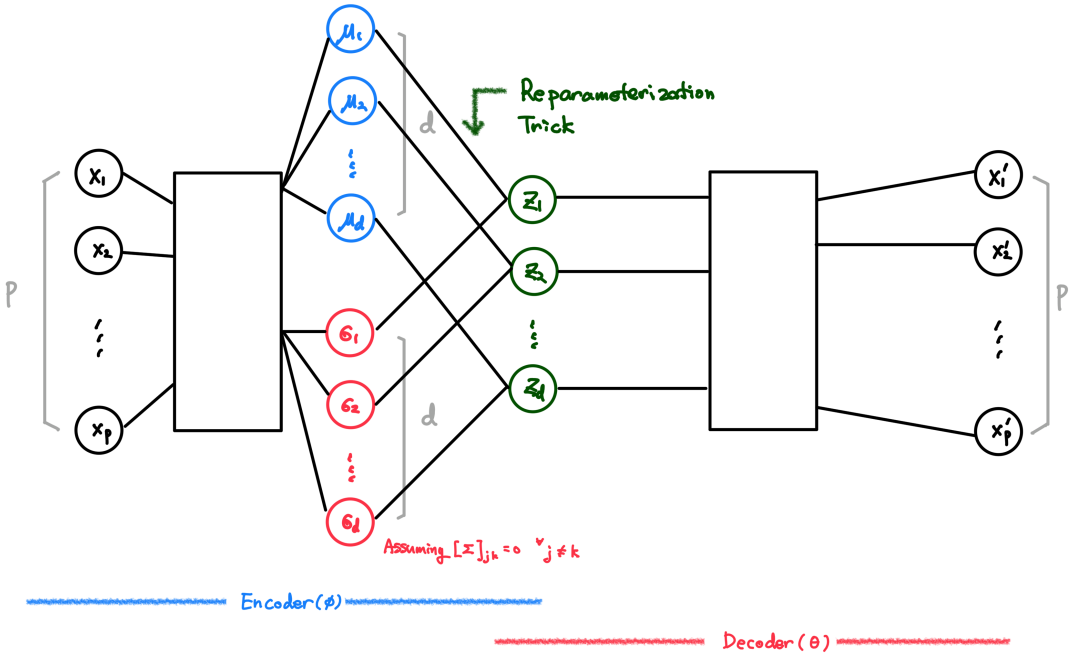


Figure 5.3: VAE as a Neural Network

Conditional VAE(CVAE) is a type of VAE that allows us to condition the generation process with labels or attributes. *It is somewhat similar to a supervised learning task*, where we have a set of input-output pairs (x_i, y_i), and we want to learn a mapping from the input x_i to the output y_i. For example, we can have portraits of people and the labels of their gender, or an MNIST datset with the labels of the digits.

5.3.1 The Problem

Consider the following decomposition of *the log-likelihood of the label given the features* wrt a data point (x_i, y_i):

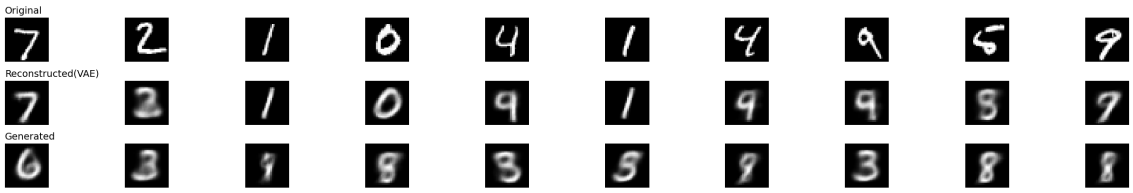


Figure 5.4: An example of MNIST reconstruction and generation with VAE

$$
\begin{aligned}
\log p_\theta(y_i|x_i) &= \log \int p_\theta(y_i, z|x_i) dz \\
&= \log \int \frac{p_\theta(y_i, z|x_i)}{q_\phi(z|x_i, y_i)} q_\phi(z|x_i, y_i) dz \\
&\geq \int \log \frac{p_\theta(y_i, z|x_i)}{q_\phi(z|x_i, y_i)} q_\phi(z|x_i, y_i) dz \qquad (5.6) \\
&= \int \log \frac{p_\theta(y_i|x_i, z) p_\theta(z|x_i)}{q_\phi(z|x_i, y_i)} q_\phi(z|x_i, y_i) dz \qquad (5.7) \\
&= \int \log p_\theta(y_i|x_i, z) q_\phi(z|x_i, y_i) dz + \int \log \frac{p_\theta(z|x_i)}{q_\phi(z|x_i, y_i)} q_\phi(z|x_i, y_i) dz \\
&= \mathbb{E}_{z \sim q_\phi(z|x_i, y_i)} \Big[\log p_\theta(y_i|x_i, z) \Big] - D_{KL} \big[q_\phi(z|x_i, y_i) || p_\theta(z|x_i) \big] \qquad (5.8)
\end{aligned}
$$

The inequality 5.6 is due to Jensen's inequality, and the equality 5.7 is due to the fact that $p_\theta(y_i, z|x_i) = p_\theta(y_i|x_i, z)p_\theta(z|x_i)$. In this section, we will use the same notation of parameters: ϕ denotes the encoding parameters(i.e., the recognition and understanding parameter of the data), and θ denotes the decoder parameters(i.e., the generation parameter of the data).

6 Advanced VAE Models(Wk 4-2)(TBU)

TBU

6.1 Disentanglement(TBU)

TBU

6.2 β-VAE

Sometimes, the generated figures from VAE might not be satisfactory. Indeed, both the reconstructed and the generated MNIST figures in figure 5.4 does not look ideal. One might say that the images are too blurry, or one might want to have more control over which digits to generate. **β-VAE** provides the simplest cure. It starts by modifying the loss function in equations 5.1 and 5.2 to the following:

$$L_i = -\mathbb{E}_{z\sim q_\phi(z|x_i)}\Big[\log p_\theta(x_i|z)\Big] + \beta \cdot D_{KL}\big[q_\phi(z|x_i)||p(z)\big] \tag{6.1}$$

$$\overset{Repar.}{=} -\mathbb{E}_{e\sim\mathcal{N}(0,I)}\Big[\log p_\theta\big(x_i \mid z = \mu_\phi(x_i) + \sigma_\phi(x_i)\odot e\big)\Big] + \beta \cdot D_{\mathrm{KL}}\left[q_\phi(z|x_i)\,||\,p(z)\right] \tag{6.2}$$

Changing the value of β would either put more or less emphasis on the KL divergence term. If $\beta < 1$. then the model will put more emphasis on the reconstruction loss, and the model will be closer to a standard autoencoder. If so, we will be able to obtain less blurry images since there is less randomness to the generated image. On the other hand, if $\beta > 1$, then the model will more strictly match $q_\theta(z|x_i)$ to a multivariate standard normal distribution. In this case, since the multivariate standard normal distribution has an identity matrix of covariance, it will make each axis of the latent space, z_p, to be *linearly* independent of each other.[13] This allows us to have more control over the generated images, as each axis of the latent space is assigned a specific feature of the data. This is called the **disentanglement** of the latent space. Consider the MNIST dataset again. Suppose you trained a β-VAE with $\beta > 1$ and found that:

- z_0 controls the digit identity, and z_1 controls the thickness of the stroke
- You found that
  - $z_0 = -2 \rightarrow$ digit 7
  - $z_0 = 0 \rightarrow$ digit 1
  - $z_0 = 2 \rightarrow$ digit 3
- You found that higher z_1 means a thicker stroke

With such a procedure, we can generate images with more control at the cost of the reconstruction loss and more blurriness. In short,

TBU from page 64(68 in total note). Also write a table here to summarize the characteristics

6.3 Factor VAE(TBU)

TBU

6.4 VQ-VAE(TBU)

TBU

6.5 Tight Lowerbounds and Importance Weighted Autoencoders(TBU)

TBU

[13] Indeed, it must be noted that the identity covariance matrix only ensures linear independence, not the independence in general.

7 Autoregressive Models(WK 5-1)

Autogregressive models are a class of generative models that generate data sequentially, one step at a time, by conditioning on the previously generated data. They originate from the literature on time-series models, where observations from the previous time-steps are used to predict the value of the current (or future) observation. For example, a specific stock price at time t can be modeled as:

$$y_t = c + w_1 y_{t-1} + w_2 y_{t-2} + \cdots + w_p y_{t-p} + \epsilon_t$$
$$\epsilon_t \sim \mathcal{N}(0, \sigma^2)$$

As in variational inference, we want to obtain $p_\theta(\mathbf{x})$, the joint distribution of the data, configured by θ. However, the joint distribution is not tractable in general; autoregressive models' key idea is to decompose the joint distribution into a chain of conditional distribution:

$$p_\theta(\mathbf{x}) = p_\theta(x_1)p_\theta(x_2|x_1)\cdots p_\theta(x_p|x_{p-1}, \cdots, x_1) = \prod_{i=1}^{p} p_\theta(x_i|x_{<i}) \tag{7.1}$$

where p denotes the dimension of the input. Note that each of these conditional distributions will be more tractable than the joint distribution, and it is generally much more tractable in the earlier time steps than in the later time steps. However, one drawback of autoregressive models is that they can be slow; they are not parallelizable, as the generation of each data point depends on the previously generated data points. In this section, we will discuss what autoregressive models are, how they are implemented in image, text, and audio data, and how their performance can be improved with different variations.

7.1 Autoregressive Models and Likelihood

The goal of learning is to maximize the likelihood of the data over the dataset $p_{\text{data}}(\mathbf{x})$. In other words, we want to infer θ such that

$$\theta = \arg\min_\theta D_{KL}\left[p_{\text{data}}||p_\theta\right] = \mathbb{E}_{x\sim p_{\text{data}}}\left[\log p_{\text{data}}(x) - \log p_\theta(x)\right] \tag{7.2}$$
$$= \arg\max_\theta \mathbb{E}_{x\sim p_{\text{data}}}\left[\log p_\theta(x)\right] \tag{7.3}$$
$$\overset{AR}{=} \arg\max_\theta \mathbb{E}_{x\sim p_{\text{data}}}\left[\sum_{i=1}^{p} \log p_\theta(x_i|x_{<i})\right] \tag{7.4}$$

where p_{data} is the data distribution which can be considered the ground-truth, or an oracle distribution, and p_θ is the model distribution. That is, we want to find a set of parameters θ such that the model distribution p_θ is as close to the data distribution p_{data} as possible.[14]

One should know that steps 7.2 and 7.3 are common in most likelihood-based models, including VAEs. However, in variational inference and variational encoders, we were not directly optimizing the log likelihood, $\log p_\theta(x)$, but rather the **evidence lower bound(ELBO)**, by minimizing the reverse KL between the

[14]Note that the KL divergence here can be considered a forward KL, because the oracle distribution comes before the model distribution. Recall that in variational inference, we had reverse KL, where we had the model distribution as the oracle and the variational distribution as the one we can optimize.

variational distribution and the model distribution. On the other hand, in autoregressive models, we are directly optimizing the log likelihood of the data, $\log p_\theta(x)$ by introducing some 'simplicity' to the data generating process: observation in t depends only on the previous observations until $t-1$.

7.2 Fully Visible Sigmoid Belief Network

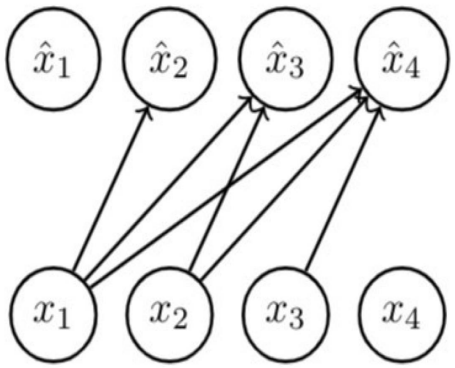


Figure 7.1: Fully visible sigmoid belief network

Fully visible sigmoid belief network(FVSBN) is an early model in AR to model **Bernoulli** random variables(e.g., black and white pixels). The name itself implies that there are no hidden layers(*fully visible*), and we use a sigmoid function to model the expected value of the Bernoulli random variable(i.e., the probability). As in figure 7.1, each Bernoulli random variable, x_t, depends on $x_{<t}$, which is then modelled by the sigmoid activation function, σ. That is,

$$\hat{x}_t = p(x_t = 1 | x_{<t}) = \sigma\big(w_0^{(t)} + w_1^{(t)} x_1 + \cdots + w_{t-1}^{(t)} x_{t-1}\big) \tag{7.5}$$

In other words, FVSBN is *equivalent to a logistic model*, where the weights are distinct for each x_t. Here, the number of parameters is $1+2+\cdots+n \in O(p^2)$, where p is the number of Bernoulli random variables:[15]

$$W_{\text{fvsbn}} = \begin{pmatrix} w_0^{(1)} & 0 & 0 & \cdots & 0 \\ w_0^{(2)} & w_1^{(2)} & 0 & \cdots & 0 \\ Gw_0^{(3)} & w_1^{(3)} & w_2^{(3)} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_0^{(p)} & w_1^{(p)} & w_2^{(p)} & \cdots & w_{p-1}^{(p)} \end{pmatrix} \tag{7.6}$$

This is much better than considering all the possible combinations of $x_1, \ldots, x_p$, which is 2^p, but still not ideal for high-dimensional data. This is where **Neural Autoregressive Distribution Estimator(NADE)** comes in, which is discussed in the next section.

[15]Notation of Big-O is described in appendix A.1 in detail.

7.3 Neural Autoregressive Distribution Estimator(NADE)

Neural Autoregressive Distribution Estimator(NADE) improves FVSBN with two features:

1. **Hidden Layer**: It introduces a hidden layer between the input and output layers, which allows for more complex interactions between the input variables.[16]

2. **Parameter Sharing**: NADE uses a shared weight matrix for all the Bernoulli random variables, x_t, instead of having distinct weights for each x_t. This reduces the number of parameters significantly.

First, we are going to have a $d(< p)$ dimensional hidden layer TBU
The shared matrix W_{nade} can be defined as equation 7.7. Comparing

$$W_{\text{nade}} = \begin{pmatrix} w_0 & 0 & 0 & \cdots & 0 \\ w_0 & w_1 & 0 & \cdots & 0 \\ w_0 & w_1 & w_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_0 & w_1 & w_2 & \cdots & w_{p-1} \end{pmatrix} \tag{7.7}$$

7.4 Masked Autoencoder Estimator(MADE)(TBU)

TBU

7.5 PixelCNN(TBU)

TBU

7.6 PixelRNN - Row LSTM(TBU)

TBU

7.7 PixelRNN - Diagonal BiLSTM(TBU)

TBU

7.8 Gated PixelCNN(TBU)

TBU

7.9 AR for Audio and Text Data(TBU)

TBU

7.9.1 Wavenets(TBU)

TBU

[16] NADE can indeed have multiple hidden layers, which is true in deep NADE and Masked Autoencoder Estimator(MADE). In this section, however, we will only discuss the original version of NADE, where we have only a single hdden layer.

7.9.2 Tokenizers(TBU)

TBU

7.10 Topics in Transformers

TBU

8 Advanced AR Models(Wk 5-2)(TBU)

8.1 Parti(TBU)

TBU

8.2 Distribution Augmentation(DistAug) (TBU)

TBU

9 Flow Based Models(Wk 6-1)

9.1 Normalizing Flows

Normalizing flows are a class of generative models that learn to transform a simple **base distribution** (like a Gaussian) into a more complex distribution through a series of invertible transformations.

That is, we put a base distribution $p(u)$ through an invertible transformation $f : \mathbb{R}^D \to \mathbb{R}^D$:

$$\begin{cases} x = f(u) \\ u \sim p(u) \end{cases} \tag{9.1}$$

The complex distribution, often called the **target distribution**, can be expressed as a composition of these transformations.

Using the **change-of-variables formula**, the probability density function of the target distribution $p(x)$ can be computed as:

$$p_X\big(x\big) = p_U\big(f^{-1}(x)\big) \cdot \big|\det \mathbf{J}\big(f^{-1}(x)\big)\big| = p_U\big(f^{-1}(x)\big) \cdot \big|\det \mathbf{J}\big(f(u)\big)\big|^{-1} \tag{9.2}$$

where $\mathbf{J}\big(f(u)\big)$ is the Jacobian of the transformation f at point u:

$$\mathbf{J}\big(f(u)\big) = \frac{\partial f(u)}{\partial u} = \begin{bmatrix} \frac{\partial f_1(u)}{\partial u_1} & \cdots & \frac{\partial f_1(u)}{\partial u_D} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_D(u)}{\partial u_1} & \cdots & \frac{\partial f_D(u)}{\partial u_D} \end{bmatrix} \in \mathbb{R}^{D \times D} \tag{9.3}$$

10 Generative Adversarial Networks (GANs)

10.1 Introduction

So far, we have discussed generative models that explicitly model the data distribution $p(x)$, such as VAEs and normalizing flows. Indeed VAEs define intractable density function with latent variable z, which lead to optimizing the evidence lower bound(ELBO) rather than directly maximizing the likelihood of the data. However, there are other approaches to generative modeling that do not require explicit modeling of the data distribution. These approaches are called **implicit generative models**. One popular implicit model is the **generative adversarial networks** (GANs), which we will discuss in this section.

In VAEs, recall that we optimized $q(z|x)$ so that we could not only sample from a simple distribution $p(z)$, but also encourage the latents to have more information about the data x. However, this restricted us from being able to sample from complex and high-dimensional distributions. GAN resolves this issue by first sampling from a simple distribution (like Gaussian or uniform), $p(z)$,and then the **generator** $G(z) : \mathbb{R}^d \to \mathbb{R}^p$ maps the latent variable z to a more complex training distribution of data x. (Note that d is the dimension of the latent space and p is the dimension of the data space.)

RMK: It is not the latent distribution that becomes complex, but the data distribution $p(x)$, or more rigorously the generated $p(\hat{x})$ that becomes complex.

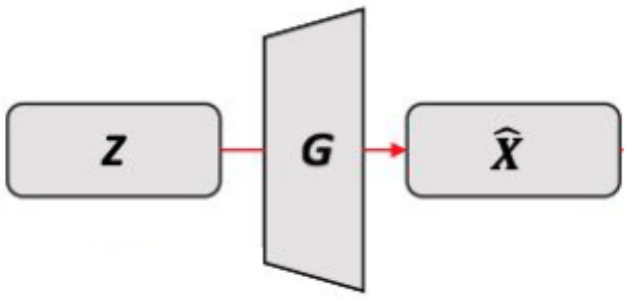


Figure 10.1: The generator neural network of GAN

After the generation process, however, the generated data $\hat{x}$ is not guaranteed to be similar to the training data x. This is where the **discriminator** $D(x) : \mathbb{R}^p \to [0, 1]$ comes in. The discriminator is a binary classifier that takes in either the real data x or the generated data $\hat{x}$ and outputs a probability of whether the input is real or fake using a sigmoid activation function.

Indeed, the process of GAN can be viewed as a two-player game between the generator and the discriminator, where one tries to fool the other.

Specifically, the loss functions of the generator and discriminator are defined as follows:

$$\mathcal{L}_D = -\mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] - \mathbb{E}_{z\sim p(z)}[\log(1 - D(G(z)))] \tag{10.1}$$

$$\mathcal{L}_G = \mathbb{E}_{z\sim p(z)}[\log(1 - D(G(z)))] \tag{10.2}$$

where $p_{data}(x)$ is the distribution of the training data, $p(z)$ is the distribution of the latent variable, and

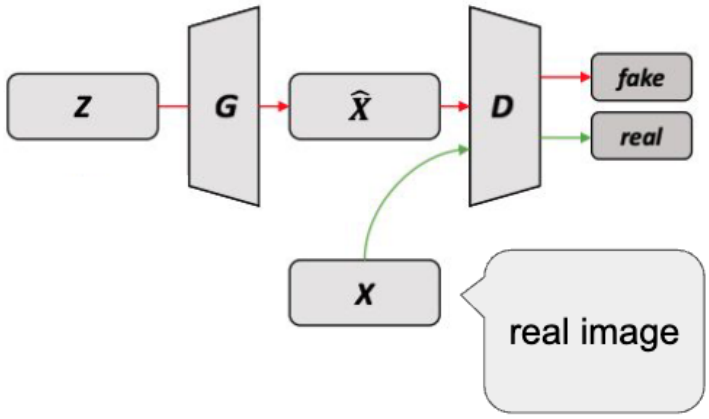


Figure 10.2: Full process of GAN

$G(z)$ is the generated data from the generator.

Since both terms in equation 10.1 are negative, we can consider the minimization of the loss function as the maximization of the inverse of each term. Then, maximizing $\mathbb{E}_{x\sim p_{data}(x)}[\log D(x)]$ is equivalent to having discriminator D assign higher probability values to the real training data x; maximizing $\mathbb{E}_{z\sim p(z)}[\log(1 - D(G(z)))]$ is equvalent to minimzing the assigned probability values to the generated data $\hat{x}$. Thus, the discriminator is trained to maximize the probability of correctly classifying the real and generated data.

On the other hand, the generator is trained to maximize the probability of the discriminator being fooled by the generated data $\hat{x}$. That is captured by maximizing $\mathbb{E}_{z\sim p(z)}[\log D(G(z))]$.

Compare equation 10.2 to the second term of the discriminator loss function in equation 10.1. Although these two terms are not interchangeable, they are closely related: equation 10.2 represents the log probability that the discrimminator is fooled, and the second term of equation 10.1 is the log probability that the discriminator rightly rejects the generated data. Based on this heuristic, Goodfellow (2014) proposes the following minimax game in place of the loss functions:[17]

$$\min_G \max_D \mathcal{L}(G, D) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p(z)}[\log(1 - D(G(z)))] \tag{10.3}$$

10.2 What is the optimization result?

Equation 10.3 is indeed the key to understading GANs. However, the form of this functionstill does not tell anything about the relationship between the distribution of the generated data, $p(\hat{x})$, and the distribution of the training data, $p_{data}(x)$. In this section, we will show that the optimization of the minimax game in equation 10.3 leads to the generator producing samples from the same distribution as the training data, provided that our discriminator is optimal.

10.2.1 Finding the optimal discriminator

Denote the minimax function by $V(G, D) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p(z)}[\log(1 - D(G(z)))]$.

Then,

[17]The need of this transformation was required because when generators and discriminators are built with neural networks and equation 10.1 is not convex, learning in GANs tend to be challenging.

$$V(G,D) = \int_x p_{data}(x)\log D(x)dx + \int_z p(z)\log(1-D(G(z)))dz \tag{10.4}$$

$$= \int_x p_{data}(x)\log D(x)dx + \int_x p_g(x)\log(1-D(x))dx \tag{10.5}$$

The equivalence of $\int_z p(z)\log(1-D(G(z)))dz$ and $\int_x p_g(x)\log(1-D(x))dx$ in equations 10.4 and 10.5 stems from the **Dirac delta sifting property**:[18]

$$\int_z p_z(z)\log\Big(1-D(G(z))\Big)dz = \int_z p_z(z)\Big[\int_x \delta\big(x-G(z)\big)\log(1-D(x))dx\Big]dz \text{ (by the sifting property)} \tag{10.6}$$

$$\left(\because f(G(z)) = \int_{-\infty}^{\infty} f(x)\delta(x-G(z))dx \text{ for any continuous} f\right)$$

$$= \int_x \int_z p_z(z)\delta(x-G(z))dz\log(1-D(x))dx \tag{10.7}$$

$$= \int_x p_z(G^{-1}(x))\log(1-D(x))dx \tag{10.8}$$

$$= \int_x p_g(x)\det\left(\frac{dG^{-1}}{dx}\right)^{-1}\log(1-D(x))dx \tag{10.9}$$

For now, we will assume that the Jacobian determinant is equal to 1(i.e., $\det\left(\frac{dG^{-1}}{dx}\right)^{-1} = 1$)(TBU)
Then, from equation 10.5, we have the following FOC wrt D:

$$V(G,D)' = \frac{p_{data}(x)}{D(x)} - \frac{p_g}{1-D(x)} = 0 \tag{10.10}$$

and thus, the optimal discriminator is given by:

$$D^*(x) = \frac{p_{data}(x)}{p_{data}(x)+p_g(x)} \tag{10.11}$$

Plugging in D^* to the minimax function,

[18] For details of the property, see appendix F.1.

$$
\begin{aligned}
V(G, D^*) &= \int_x p_{data}(x) \log D^*(x) dx + \int_x p_g(x) \log(1 - D^*(x)) dx \\
&= \mathbb{E}_{x \sim p_{data}}[\log D^*(x)] + \mathbb{E}_{x \sim p_g}[\log(1 - D^*(x))] \\
&= \mathbb{E}_{x \sim p_{data}}[\log \frac{p_{data}(x)}{p_{data}(x) + p_g(x)}] + \mathbb{E}_{x \sim p_g}[\log \frac{p_g(x)}{p_{data}(x) + p_g(x)}] \\
&= \mathbb{E}_{x \sim p_{data}}\left[\log \frac{p_{data}(x)}{\frac{p_{data}(x) + p_g(x)}{2}}\right] - \log 2 + \mathbb{E}_{x \sim p_g}\left[\log \frac{p_g(x)}{\frac{p_{data}(x) + p_g(x)}{2}}\right] - \log 2 \\
&+ 2 \cdot D_{JS}(p_{data} || p_g) - log4
\end{aligned}
$$

where $D_{JS}(p_{data}||p_g)$ is the **Jensen-Shannon divergence** between the two distributions. Since $D_{JS}(p_{data}||p_g) = 0$ if and only if $p_{data}(x) = p_g(x)$, we can conclude that given the optimal discriminator D^*, our generator G's goal is to produce samples from the same distribution as the training data $p_{data}(x)$. In other words, the perfect teacher D^* nurtures the perfect student $p_g = p_{data}$.

10.3 Gradient ascent and descent

For minimization and the maximization of equation 10.3, we can use the **stochastic gradient descent** (SGD) algorithm:

Minimax objective	$\min_{\theta_g} \max_{\theta_d} \mathbb{E}_{x \sim p_{data}(x)}[\log D_{\theta_d}(x)] + \mathbb{E}_{z \sim p(z)}[\log(1 - D_{\theta_d}(G_{\theta_g}(z)))]$
ascent on the discriminator	$\max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}}[\log D_{\theta_d}(x)] + \mathbb{E}_{z \sim p(z)}[\log(1 - D_{\theta_d}(G_{\theta_g}(z)))]\right]$
descent on the generator	$\min_{\theta_g} \left[\mathbb{E}_{z \sim p(z)}[\log(1 - D_{\theta_d}(G_{\theta_g}(z)))]\right]$

Consider the descent on the generator. The first-order condition of the generator is given by:

$$
\begin{aligned}
\nabla_{\theta_g} \mathcal{L}_G &= \nabla_{\theta_g} \mathbb{E}_{z \sim p(z)}[\log(1 - D_{\theta_d}(G_{\theta_g}(z)))] \\
&= \nabla_{\theta_g} \int_z \log(1 - D_{\theta_d}(G_{\theta_g}(z))) p(z) dz \\
&= \int_z \nabla_{\theta_g} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) p(z) dz \\
&= -\int_z \frac{1}{1 - D_{\theta_d}(G_{\theta_g}(z))} \nabla_{\theta_g} D_{\theta_d}(G_{\theta_g}(z)) p(z) dz \\
&= -\int_z \frac{1}{1 - D_{\theta_d}(G_{\theta_g}(z))} \nabla_x D_{\theta_d}(G_{\theta_g}(z)) \nabla_{\theta_g} G_{\theta_g}(z) p(z) dz \\
&= -\mathbb{E}_{z \sim p(z)}\left[\frac{1}{1 - D_{\theta_d}(G_{\theta_g}(z))} \nabla_x D_{\theta_d}(G_{\theta_g}(z)) \nabla_{\theta_g} G_{\theta_g}(z)\right]
\end{aligned}
$$

But consider the case when the generator is optimal, deceiving the discriminator perfectly(i.e., $D_{\theta_d}(G_{\theta_g}(z)) = 1$ for all z). Then, $\frac{1}{1 - D_{\theta_d}(G_{\theta_g}(z))} \to \infty$, sending the gradient to $-\infty$. To prevent the gradient from exploding, we use **non-saturating loss** for the generator in practice[19], which is given by:

[19] This explanation is from our class lecture. However, as I searched for more information through ChatGPT, I found another, perhaps more compelling, reason for using the non-saturating loss. Please check it out in

$$\mathcal{L}_G = -\mathbb{E}_{z\sim p(z)}[\log D_{\theta_d}(G_{\theta_g}(z))] \tag{10.12}$$

The goal would be to maximize the probability of the discriminator being fooled by the generated data $\hat{x}$:

$$\max_{\theta_g} \mathbb{E}_{z\sim p(z)}[\log D_{\theta_d}(G_{\theta_g}(z))] \text{ (\textbf{ascent} on the generator)} \tag{10.13}$$

Note that we cannot simply replace the second term of equation 10.3 with 10.12 because it hinders the optimization of the discriminator:

$$\max_G \max_D \mathcal{L}(G, D) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p(z)}[\log D(G(z))] \tag{10.14}$$

That is, in equation 10.14, the discriminator is trained to maximize both the probability of generated data being accepted and real data being accepted. This removes the adversarial tension, and thus we have to separate the optimization of the two players:

$$\max_{\theta_d} -\mathcal{L}_D = \mathbb{E}_{x\sim p_{data}(x)}\left[\log D_{\theta_d}(x)\right] + \mathbb{E}_{z\sim p(z)}\left[\log(1 - D_{\theta_d}(G_{\theta_g}(z)))\right] \tag{10.15}$$

$$\max_{\theta_g} -\mathcal{L}_G = \mathbb{E}_{z\sim p(z)}\left[\log D_{\theta_d}(G_{\theta_g}(z))\right] \tag{10.16}$$

To visualize, see the different gradient for what is inside the expectation when $D(G(z)) \to 1$ in figure 10.3.

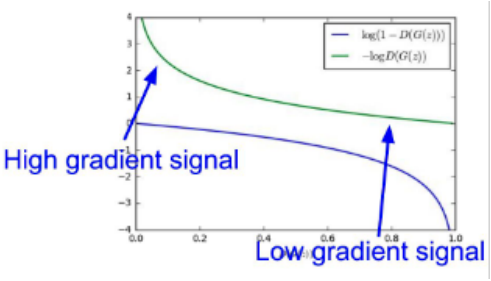


Figure 10.3: The gradient of the generator loss function

10.3.1 Why not change equation 10.15?(Personal)

This part of the document was not discussed in the lecture, but I added it from ChatGPT for my own understanding.

In equation 10.15, we can see that we do not need a fix for the discriminator for non-saturating loss. The discriminator's loss is:

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

This is a standard binary cross-entropy loss for classifying real versus fake samples and is well-behaved:

1. Ground-truth labels are known (`real` $= 1$, `fake` $= 0$). Indeed, it is a supervised classification problem.
2. The loss landscape for cross-entropy is well-behaved, even when predictions are close to 0 or 1.
3. Most importantly:
   - If $D(G(z)) \to 0$, then $\log(1 - D(G(z))) \to 0$, but
   - The gradient is:
   $$\frac{d}{dD} \log(1 - D) = -\frac{1}{1 - D}$$
   which does not vanish as long as $D < 1$. In fact, it diverges as $D \to 1$.

10.4 Final Results

The distribution of the generated data $p_g(x)$, the true distribution of the actual data $p_{data}(x)$, and the probability of the discriminator accepting the data $p_D(x)$, can be visualized as in figure 10.4.

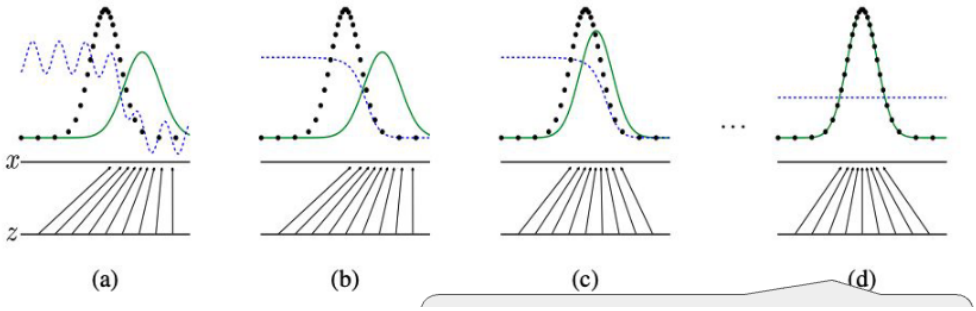


Figure 10.4: Optimization process of GAN

In figure 10.4, the blue dashed line represents $p_D(x)$, the green solid line represents $p_g(x)$, and the **black dotted line** represents $p_{data}(x)$. As the stages progress, the generator G learns to produce samples that are more similar to the training data $p_{data}(x)$, making it more difficult for the discriminator D to distinguish between real and generated data.

10.5 VAE/GAN

VAE/GAN is a hybrid model that combines the strengths of both VAEs and GANs. The VAE part of the model is used to learn a latent representation of the data, while the GAN part is used to generate high-quality samples from that latent representation. That is, the GAN discriminator works as the basis for the VAE reconstruction loss, which replaces the pixel-wise error with feature-wise error.[20] **Important Note:** This does not mean that the GAN discriminator fully repalces the VAE's reconstruction loss. They work as a separate pair of teachers: the VAE reconstruction loss is used to train the encoder, while the GAN discriminator is used to train the decoder.

TBU

[20] Notice that GAN does not evaluate an image based on the pixel-wise error, but rather on the plausibility of the entire image.

	Reconstruction Loss(Encoder)	Generation Loss(Decoder)	Discriminator Loss
VAE	$\mathbb{E}_{z\sim q_\phi(z\|x_i)}\left[\log p_\theta(x_i\|z)\right]$	$D_{KL}[q_\phi(z\|x_i)\|\|p(z)]$	
GAN		$\mathbb{E}_{z\sim p(z)}\left[\log D(G(z))\right]$	$-\mathbb{E}_{x\sim p_{data}(x)}[\log D(x)]$

Table 2: Comparison of VAE and GAN

Notice that in figure 5.3, the VAE decoder was a NN that mapped the latent variable z to the data space x. In that aspect, the VAE decoder is similar to the generator of GAN. The difference of the two stems from the difference in the loss function. In VAE/GAN, however, we are using the loss function of GAN to train the decoder; thus, the decoder can be viewed as a generator from now on. Specifically, we use the decoder as in the VAE and generate the data $\hat{x}$ from the latent variable $z \sim \mathcal{N}(0, \mathbf{I})$. Then, on top of the reconstruction loss term in table 2, we add the GAN discriminator to evaluate the generated data $\hat{x}$ and the training data x.

The algorithm is as follows:

Algorithm 10.1 VAE/GAN

Input: Training data x_i, latent variable z_i, encoder $q_\phi(z|x)$, decoder $p_{\theta_1}(x|z)$, discriminator $D_{\theta_2}(x)$

Output: Generated data $G(z)$

$\phi, \theta_1, \theta_2 \leftarrow$ initialize network parameters(encoder, decoder, discriminator)

repeat

$\mathbf{X} \leftarrow$ random batch from training data

$\mathbf{Z} \leftarrow \text{Enc}_\phi(\mathbf{X})$

Compute $\mathcal{L}_{\text{prior}} \leftarrow D_{KL}[q_\phi(z|x)||p(z)]$

$\tilde{\mathbf{X}} \leftarrow \text{Dec}_{\theta_1}(\mathbf{Z})$

Compute $\mathcal{L}_{\text{recon}} \leftarrow -\mathbb{E}_{z\sim q_\phi(z|x)}\left[\log p_{\theta_1}(x|z)\right]$

$\mathbf{Z}_p \leftarrow$ samples from the prior $p(z) \sim \mathcal{N}(0, \mathbf{I})$

$\tilde{\mathbf{X}}_p \leftarrow \text{Dec}_{\theta_1}(\mathbf{Z}_p)$

Compute $\mathcal{L}_{\text{GAN}} \leftarrow \log(D_{\theta_2}(\mathbf{X})) + \log(1 - D_{\theta_2}(\tilde{\mathbf{X}})) + \log(1 - D_{\theta_2}(\mathbf{X}_p))$

// Update parameters according to the gradients

$\phi \leftarrow \phi - \nabla_\phi \mathcal{L}_{\text{prior}} - \nabla_\phi \mathcal{L}_{\text{recon}}$

$\theta_1 \leftarrow \theta_1 - \gamma \cdot \nabla_{\theta_1} \mathcal{L}_{\text{recon}} + \nabla_{\theta_1} \mathcal{L}_{\text{GAN}}$

$\theta_2 \leftarrow \theta_2 - \nabla_{\theta_2} \mathcal{L}_{\text{GAN}}$

until convergence

10.6 Adversarial Autoencoders(AAE)

In standard VAEs, the encoder $q_\phi(z|x)$ is trained to match the prior distribution $p(z)$. Such prior distributions were usually chosen to be simple distributions, such as Gaussian or uniform. However, in some cases, we may want to learn a more complex prior distribution that better captures the structure of the data. In sections 10.6 and 10.7, we will discuss two methods to learn a more complex prior distribution: **adversarial autoencoders** (AAEs) and **adversarial variational Bayes** (AVB).

Adversarial autoencoders (AAEs) are a type of generative model that combines the ideas of VAEs and GANs to learn a more complex prior distribution. Unlike traditional autoencoders, which focus primarily on reconstructing input data, AAEs aim to match the latent space to a complex predefined prior distribution by adding an adversarial training process.

Consider why we could not use complex prior distributions in VAEs. IN VAEs, we relied on the KL divergence to measure the difference between the learned distribution and the prior distribution. For the KL divergence to be computed efficiently and analytically, the prior ($p(z)$) must have a closed-form expression and be easy to sample from. Complex prior distributions, such as multimodal or highly structured distributions, often lack these properties, making the KL divergence intractable. This limitation prevents VAEs from directly using complex priors that could better capture the underlying structure of the data.

Adversarial Autoencoders (AAEs) address this limitation by *replacing the KL divergence with an adversarial training process*, allowing the model to learn more complex prior distributions without requiring a closed-form KL divergence.

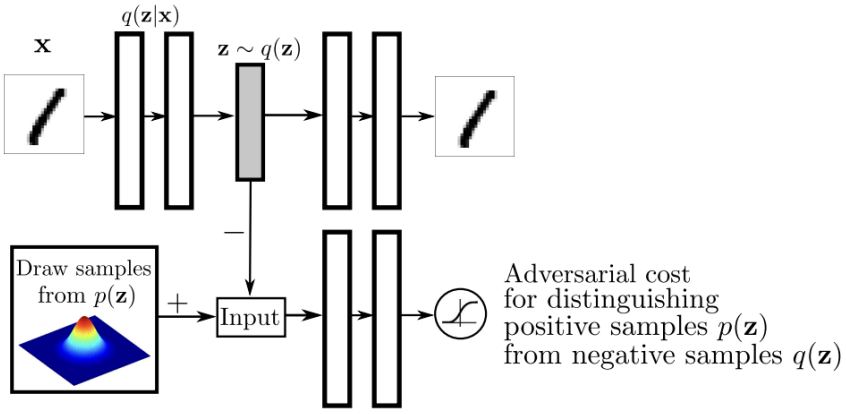


Figure 10.5: Adversarial Autoencoders

In figure 10.5, the first row shows the standard autoencoder architecture *without any consideration of the prior distribution* and thus without the KL divergence. It is important to note that this is an 'autoencoder', not a VAE. That is, distribution parameterization(μ, Σ) is not considered, but we can compute the resulting distribution of latent variables, $q(z|x)$ and $q(z)$:

$$q(z) = \int_X q(z|x)p_{data}(x)dx$$

Buy for generation, we need the prior distribution $p(z)$ to sample from. Since we set a complex prior distribution, instead of using the KL divergence to compare $p(z)$ and $q(z)$, we use the GAN discriminator to compare the two distributions. - This step is described in the second row of figure 10.5. - This is equivalent to training the encoder $q(z|x)$ to fool the discriminator $D(z)$, which is trained to distinguish between samples from the prior distribution $p(z)$ and samples from the encoder $q(z|x)$.

Moreover, one can inject label information into the latent space by conditioning the encoder and decoder on the labels. This is particularly useful in semi-supervised learning scenarios, where we have a small amount of labeled data and a large amount of unlabeled data. The way AAE incorporates the label information can

be summarized as follows:

$$
\begin{aligned}
q(z|x,y) &= \text{Enc}_\phi(x,y) \\
p(x|z,y) &= \text{Dec}_\theta(z,y) \\
p(z|y) &= \text{Prior}(y)
\end{aligned}
$$

where y is the label information. *In short, you can think of this framework as performing AAEs separately for each class y.*

10.7 Adversarial Variational Bayes(AVB)

Consider the VAE's objective function:

$$
\max_\phi \max_\theta \mathbb{E}_{x\sim p_{data}(x)}\Big[-D_{KL}\big(q_\phi(z|x)||p(z)\big) + \mathbb{E}_{z\sim q_\phi(z|x)}\Big[\log p_\theta(x|z)\Big]\Big] \tag{10.17}
$$

By expanding the KL divergence,

$$
\max_\phi \max_\theta \mathbb{E}_{x\sim p_{data}(x)}\Big[\mathbb{E}_{z\sim q_\phi(z|x)}\Big[\log p(z) - \log q_\phi(z|x) + \log p_\theta(x|z)\Big]\Big] \tag{10.18}
$$

How did we handle this objective function? That is, how did we model each term? We handled $p(z)$ and $q_\phi(z|x)$ separately and let q_ϕ be as similiar as possible to $p(z)$. Since $p(z)$ was a simple distribution like Gaussian, we could calculate the KL divergence easily if we had q_ϕ follow Gaussian as well; we did so by the reparameterization trick. Because of treating these two distributions separately, we could only model the sampling with simple distributions. The goal of **adversarial variational Bayes(AVB)** is to learn a more complex prior distribution $p(z)$ by combining the two terms and using the GAN discriminator to model the KL divergence between $q_\phi(z|x)$ and $p(z)$.

Rewrite equation 10.18 as

$$
\max_\phi \max_\theta \mathbb{E}_{x\sim p_{data}(x)}\Big[\mathbb{E}_{z\sim q_\phi(z|x)}\Big[\log \frac{p(z)}{q_\phi(z|x)} + \log p_\theta(x|z)\Big]\Big] \tag{10.19}
$$

TBU LetTBU need to fix $\sigma\big(T(x,z)\big) = \log\left(\frac{p(z)}{q_\phi(z|x)}\right)$ where σ is a sigmoid function.

Then,

References

Goodfellow, Ian J (2014). "On distinguishability criteria for estimating generative models". In: *arXiv preprint arXiv:1412.6515*.

Hogg, Robert V, Joseph W McKean, Allen T Craig, et al. (2013). *Introduction to mathematical statistics*. Pearson Education India.

APPENDIX

A Notations

A.1 Big-O and Big-Theta

A.1.1 Big-O

Definition A.1.1. *The notation $f(n) = O(g(n))$ means that there exist positive constants C and n_0 such that for all $n \geq n_0$, we have:*

$$|f(n)| \leq C|g(n)| \tag{A.1}$$

This means that $f(n)$ grows at most as fast as $g(n)$, up to a constant factor, for sufficiently large n.

Big-O notation can be thought of as a description of an **asymptotic upper bound** on the growth rate of a function. For example, suppose $f(n)$ is the number of model parameters given the size of the data, n. Then, $O(g(n))$ means that the number of model parameters *grow no faster* than $g(n)$, up to a constant factor, for sufficiently large n.

Here, $g(n)$ need not be unique. For example, $O(n^2)$ and $O(n^3)$ are both valid choices for $g(n)$ if $f(n) = n^2$, as they both satisfy the definition of Big-O.

A.1.2 Big-Θ

Definition A.1.2. *The notation $f(n) = \Theta(g(n))$ means that there exist positive constants C_1, C_2, and n_0 such that for all $n \geq n_0$, we have:*

$$C_1|g(n)| \leq |f(n)| \leq C_2|g(n)| \tag{A.2}$$

This means that $f(n)$ grows at the same rate as $g(n)$, up to constant factors, for sufficiently large n.

Big-Θ notation can be thought of as a description of aa **tight asymptotic tight bound** on the growth rate of a function. Think of Big-O as a *loose club* of functions where any function growing no faster is allowed(including even the much smaller ones). Big-Θ is an *exclusive subset* of that club, where only the functions growing exactly like $g(n)$ are allowed.

Corollary A.1.1. *If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, then $f(n) = \Theta(g(n))$.*

i.e., $\Theta(g(n))$ implies $O(g(n))$.

B Distributions

B.1 Dirichlet Distribution

The **Dirichlet distribution** (after Peter Gustav Dirichlet), often denoted $Dir(\alpha)$, represents a continuous multivariate distribution parameterized by $\alpha \in \mathbb{R}_+^K$. It is a multivariate version of the beta distribution, which is usually suited for modeling the randomness of percentages or proportions.

Definition B.1.1. *The **Dirichlet distribution** of order $K \geq 2$ with parameters $\alpha_1, \ldots, \alpha_K$ if the probability density function is given as:*

$$f(x_1, \ldots, x_K; \alpha_1, \ldots, \alpha_K) = \frac{1}{\mathrm{B}(\boldsymbol{\alpha})} \prod_{i=1}^{K} x_i^{\alpha_i - 1} (\ \textit{for}\ {}^{\forall}\mathbf{x} \in \mathcal{S}^K) \tag{B.1}$$

with the support being the standard $K-1$ simplex:

$$\mathcal{S}^K = \{\mathbf{x} | \sum_{i=1}^{K} x_i = 1 \ \textit{and}\ x_i \in [0,1]\} \tag{B.2}$$

The normalizing constant is the multivariate beta function:

$$\mathrm{B}(\boldsymbol{\alpha}) = \frac{\prod_{i=1}^{K} \Gamma(\alpha_i)}{\Gamma\left(\sum_{i=1}^{K} \alpha_i\right)} \tag{B.3}$$

B.2 Wishart Distribution

The **Wishart distribution** (after John Wishart), often denoted $W_p(\mathbf{V}, n)$, represents a generalized version of the gamma distribution to a multidimensional case, and is very often used to model the precision(i.e., the inverse of the covariance) matrix (or any other positive semidefinite matrix that serves a similar purpose). It is parameterized by $\mathbf{V}$, the covariance matrix of a p-dimensional normal with zero mean, and n, the degree of freedom. If $p = v = 1$, the same random variable follows a **chi-square distribution** with degree n.

Definition B.2.1. *The **Wishart distribution** with $\mathbf{V}$, p, and n is defined as below.*

Suppose that $G = [\mathbf{g}_1, \mathbf{g}_2, \ldots, \mathbf{g}_n]$ is a $p \times n$ matrix where $\mathbf{g}_i \sim N_p(0, \mathbf{V})^{\forall i \in \{1,\ldots,n\}}$. Then, $S = GG' = \sum_{i=1}^{n} \mathbf{g_i} \mathbf{g_i}' \sim W_p(\mathbf{V}, n)$

Here, S has the following probability density function:

$$f_{\mathbf{S}}(\mathbf{s}) = \frac{1}{2^{np/2} |\mathbf{V}|^{n/2} \Gamma_p\left(\frac{n}{2}\right)} |\mathbf{s}|^{(n-p-1)/2} e^{-\frac{1}{2}\mathrm{tr}(\mathbf{V}^{-1}\mathbf{s})} \tag{B.4}$$

The expected value of a Wishart variable would be

$$E(\mathbf{S}) = n\mathbf{V} \tag{B.5}$$

where $\mathbf{V}$ is a $p \times p$ matrix. Our interest could alternatively be in the expected covariance matrix, which is the expected value of the inverse of S. Letting $\Sigma = S^{-1}$, the expectation is

$$E(\boldsymbol{\Sigma}) = \frac{1}{n-p-1} \mathbf{V}^{-1} \text{for } n > p+1 \tag{B.6}$$

C KL Divergence

$$\begin{aligned} D_{KL}(p(x)||q(x)) &= \int_{\mathcal{X}} p(x) \log \frac{p(x)}{q(x)} dx \\ &= \int p(x) \log p(x) dx - \int p(x) \log q(x) dx \\ &= -H(p) + H(p,q) \end{aligned} \tag{C.1}$$

Note that $H(p)$ and $H(p,q)$ represents the entropy of p and cross-entropy of p and q. For more information, visit Cross-entropy.

D Mean Field VI

Question: In mean field variational inference, why is it okay to forgo the terms unrelated to the latent of interest? It seems that the optimal prior modelling(logged) is proportional to the expectation of log joint distribution. However after obtaining the log joint distribution, they mostly drop the terms unrelated to the latent of interest. Why is this okay? Isn't it that the "entire" log joint distribution is what's proportional to our best prior modelling?

Calude 3.7 Sonnet Thinking:

Mean Field Variational Inference and Dropping Irrelevant Terms

You're asking a great question about why we can drop terms unrelated to the specific latent variable we're currently optimizing in mean field variational inference.

The key insight lies in how the optimal variational distribution is derived:

The Mathematical Justification When we're finding the optimal variational distribution for a specific latent variable group z_j, we've shown that:

$$q_j^*(z_j) \propto \exp(E_{q_{k\neq j}}[\log p(x, \mathbf{z})])$$

If we decompose the joint log-probability $\log p(x, \mathbf{z})$ into:

1. Terms that involve z_j: $f(z_j, x, z_{k\neq j})$
2. Terms that don't involve z_j: $g(x, z_{k\neq j})$

Then:

$$E_{q_{k\neq j}}[\log p(x, \mathbf{z})] = E_{q_{k\neq j}}[f(z_j, x, z_{k\neq j})] + E_{q_{k\neq j}}[g(x, z_{k\neq j})]$$

The second term doesn't depend on z_j at all, so:

$$q_j^*(z_j) \propto \exp(E_{q_{k\neq j}}[f(z_j, x, z_{k\neq j})]) \times \exp(E_{q_{k\neq j}}[g(x, z_{k\neq j})])$$

Since $\exp(E_{q_{k\neq j}}[g(x, z_{k\neq j})])$ is just a constant with respect to z_j, it becomes part of the normalization constant, not affecting the proportionality.

That's why we can focus solely on terms involving the latent variable of interest and safely drop all other terms - they don't affect the functional form of the optimal distribution for that specific latent variable.

E Variational Autoencoders

E.1 Non-differentiable Sampling

If In deep learning, we use gradient-based optimization — backpropagation — to update model parameters. That only works if every operation in the computation graph is differentiable, i.e., we can compute:

$$\frac{\partial \text{output}}{\partial \text{parameters}}$$

Suppose you want to sample a random variable: $Z \sim \mathcal{N}(\mu, \sigma^2)$. This operation involves drawing a random value directly from a probability distribution you want to optimize — not computing a function whose derivative you can take. It is inherently stochastic. So there is no clean way to compute $\frac{\partial \text{output}}{\partial \text{parameters}}$. In other words, there is no well-defined gradient from the sample back to the parameters of the distribution.

Instead of sampling $Z \sim \mathcal{N}(\mu, \sigma^2)$, you rewrite it as:

$$Z = \mu + \sigma \cdot \epsilon \quad \text{with } \epsilon \sim \mathcal{N}(0, 1) \tag{E.1}$$

Now, ϵ is the only random part, and the rest is a differentiable function of μ and σ.

Consider the VAE loss function:

$$\mathcal{L}(\theta, \phi; x) = -\mathbb{E}_{Z \sim q_\phi(z|x)}[\log p_\theta(x|z)] + D_{KL}(q_\phi(z|x)||p(z)) \tag{E.2}$$

The first term is the reconstruction loss, and the second term is the KL divergence between the approximate posterior and the prior. In this case the first term is what blocks gradients to flow back without reparameterization. We sample Z, which shapes $\log p_\theta(x|z)$ assuming that θ is fixed. However, if we sample Z directly from $q_\phi(z|x)$, we cannot compute the gradient of the first term with respect to ϕ, **because we cannot take the derivative of a sampling process itself**. Thus, we sample Z using the reparameterization trick, which allows us to express the sampling process as a deterministic function of μ, σ, and a noise variable ϵ.

E.2 A full code on MNIST reconstruction and generation

```
import tensorflow as tf
from tensorflow.keras import layers, Model
import numpy as np
import matplotlib.pyplot as plt

# Load MNIST data
(x_train, _), (x_test, _) = tf.keras.datasets.mnist.load_data()
x_train = x_train.astype("float32") / 255.
x_test = x_test.astype("float32") / 255.
x_train = x_train.reshape((-1, 28 * 28))
x_test = x_test.reshape((-1, 28 * 28))

# Hyperparameters
latent_dim = 2
input_dim = 28 * 28
```

```
epochs = 20
batch_size = 128

# Encoder
class Encoder(Model):
    def __init__(self, latent_dim):
        super().__init__()
        self.fc1 = layers.Dense(512, activation='relu')
        self.fc_mean = layers.Dense(latent_dim)
        self.fc_log_var = layers.Dense(latent_dim)

    def call(self, x):
        h = self.fc1(x)
        mean = self.fc_mean(h)
        log_var = self.fc_log_var(h)
        return mean, log_var

# Reparameterization
def sample_z(mean, log_var):
    eps = tf.random.normal(shape=tf.shape(mean))
    return mean + tf.exp(0.5 * log_var) * eps

# Decoder
class Decoder(Model):
    def __init__(self):
        super().__init__()
        self.fc1 = layers.Dense(512, activation='relu')
        self.fc2 = layers.Dense(input_dim, activation='sigmoid')

    def call(self, z):
        h = self.fc1(z)
        return self.fc2(h)

# VAE
class VAE(Model):
    def __init__(self, encoder, decoder):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder

    def call(self, x):
        mean, log_var = self.encoder(x)
        z = sample_z(mean, log_var)
        x_recon = self.decoder(z)
        return x_recon, mean, log_var

# Loss function
def compute_loss(x, x_recon, mean, log_var):
    recon_loss = tf.reduce_sum(
        tf.keras.backend.binary_crossentropy(x, x_recon), axis=1
    )
    kl_loss = -0.5 * tf.reduce_sum(1 + log_var - tf.square(mean) - tf.exp(log_var), axis=1)
    return tf.reduce_mean(recon_loss + kl_loss)
```

```
# Prepare training
encoder = Encoder(latent_dim)
decoder = Decoder()
vae = VAE(encoder, decoder)
optimizer = tf.keras.optimizers.Adam()

# Training step
@tf.function
def train_step(x):
    with tf.GradientTape() as tape:
        x_recon, mean, log_var = vae(x)
        loss = compute_loss(x, x_recon, mean, log_var)
    gradients = tape.gradient(loss, vae.trainable_variables)
    optimizer.apply_gradients(zip(gradients, vae.trainable_variables))
    return loss

# Training loop
train_dataset = tf.data.Dataset.from_tensor_slices(x_train).shuffle(60000).batch(batch_size)

for epoch in range(epochs):
    for x_batch in train_dataset:
        loss = train_step(x_batch)
    print(f"Epoch {epoch + 1}, Loss: {loss.numpy():.2f}")

# Visualize reconstruction and generation
def plot_reconst_n_generation(model, x_test):
    x = x_test[:10]
    x_recon, _, _ = model(x)
    x = x.reshape(-1, 28, 28)
    x_recon = x_recon.numpy().reshape(-1, 28, 28)

    z = tf.random.normal(shape=(10, latent_dim))  # latent_dim must match your VAE's setting
    generated = model.decoder(z)
    generated = tf.reshape(generated, [-1, 28, 28])

    plt.figure(figsize=(30, 4))

    for i in range(10):
        # Original
        ax = plt.subplot(3, 10, i + 1)
        plt.imshow(x[i], cmap="gray")
        ax.axis("off")
        if i == 0:
            ax.set_title("Original", fontsize=14, loc='left')

        # Reconstructed
        ax = plt.subplot(3, 10, i + 11)
        plt.imshow(x_recon[i], cmap="gray")
        ax.axis("off")
        if i == 0:
            ax.set_title("Reconstructed(VAE)", fontsize=14, loc='left')

        # Generated
        ax = plt.subplot(3, 10, i + 21)
```

```
            plt.imshow(generated[i], cmap='gray')
            ax.axis('off')
            if i == 0:
                ax.set_title("Generated", fontsize=14, loc='left')

        plt.tight_layout()
        plt.show()

plot_reconst_n_generation(vae, x_test)
```

F Generative Adversarial Networks

F.1 Dirac delta sifting property

Theorem F.1.1 (Dirac delta sifting property)**.** *Let $f(x)$ be a continuous function and $\delta(x)$ be the Dirac delta function. Then, we have*

$$\int_{-\infty}^{\infty} f(x)\delta(x-a)dx = f(a) \tag{F.1}$$

for any a.

Proof. The Dirac delta function $\delta(x-a)$ is defined such that it is zero everywhere except at $x=a$, where it is infinite, and its integral over the entire real line is equal to 1. Therefore, when we integrate $f(x)$ multiplied by $\delta(x-a)$, we are effectively "sifting out" the value of $f(x)$ at $x=a$.

$$\int_{-\infty}^{\infty} f(x)\delta(x-a)dx = \int_{-\infty}^{\infty} f(x)\cdot 0dx + \int_{-\infty}^{\infty} f(a)\cdot\delta(x-a)dx \tag{F.2}$$

$$= 0 + f(a)\cdot 1 = f(a) \tag{F.3}$$

Thus, the property holds. □

F.2 Non-saturating Loss

Figure 10.3 deemonstrates the comparison between the non-saturating loss and the original GAN loss. In the body, I mentioned that the non-saturating loss is needed to avoid the *exploding gradient* problem when $D(G(z))$ reaches one. However, further discussions with ChatGPT told me that the non-saturating loss is for *vanishing gradient* problem when $D(G(z))$ reaches zero(i.e., the discriminator is too good). Notice that early in the training process, the generator is not good enough to fool the discriminator, and thus $D(G(z))$ is close to zero.

The original GAN loss gradient is given as:

$$-\mathbb{E}_{z\sim p(z)}\left[\frac{1}{1-D_{\theta_d}(G_{\theta_g}(z))}\nabla_x D_{\theta_d}(G_{\theta_g}(z))\nabla_{\theta_g}G_{\theta_g}(z)\right]$$

In this case, $\left[\frac{1}{1-D_{\theta_d}(G_{\theta_g}(z))}\right]_{D(G)=0} = 1$ is not a problem. However, $\nabla_x D_{\theta_d}(G_{\theta_g}(z))$ is close to zero: gradient vanishes and the generator cannot learn anything. Thus, the non-saturating loss is used to avoid the vanishing gradient problem when $D(G(z))$ is close to zero.

The question is "why is the original GAN loss exploding when $D(G(z))$ is close to one is not a problem?" The answer is that $D(G(z)) \approx 1$ happens in the later stage of training. In this case, the generator is already good enough to fool the discriminator, and thus the gradient of the generator is not needed.