扩散模型

齐天博

tianboqi.github.io

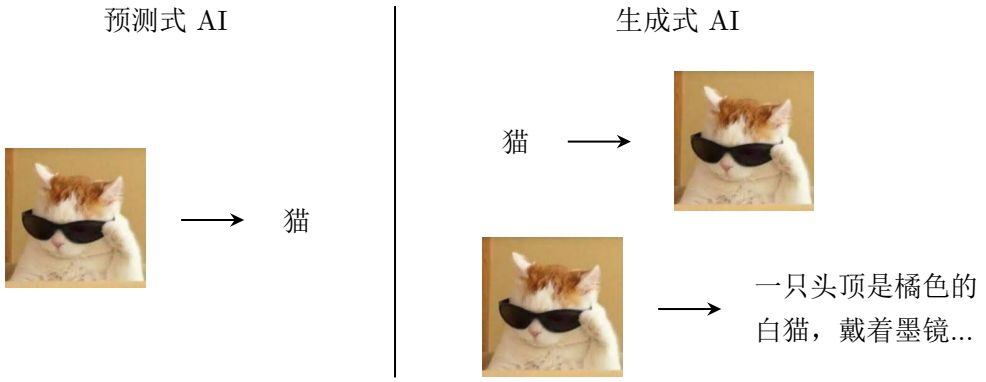
目录

1　生成式模型的基本概念

1.1　什么是生成式模型

本笔记开始写于 2024 年，现在人工智能领域在大众中最火的两个概念就是生成式 AI (generative AI) 和大语言模型 (large languge model, LLM)．前者包括著名的 AI 画图平台 DALL·E、MidJourney 和 Stable Diffusion 等，而后者则包括几乎人人在用的聊天软件 chatGPT．而实际上，LLM 也就是一类语言的生成式模型．因此可以说，这个时代是生成式 AI 的时代．

那么什么是生成式模型呢？顾名思义，生成式模型就是生成数据的模型．在这个时代之前，大多数 AI 模型都是输入一个复杂的数据，输出某种简单的判断，例如输入一张图片，判断图片里是一只猫还是一只狗．这类模型称为 **预测式模型** (predictive models)．而 **生成式模型** (generative models) 则正好相反，它的目标是生成复杂的数据，例如图像，抑或是一段复杂的文字．这种任务的复杂程度远高于预测式模型．



在这个笔记中，我们把目光局限在 **图像生成** (image generation) 任务上．图像生成是生成式模型在计算机视觉领域最主要的应用，其基本任务就是根据已有的图像数据集生成新图像．除了生成一整个图像以外，图像生成模型还可以用于一些类似的任务中，例如图像去模糊/去噪 (deblurring/denoising)、图像修复 (inpainting)、图像着色 (colorization)、风格转换 (style transfer)、根据 2D 图像生成 3D 图像等．

得益于算力的提升和算法的开发，生成式模型在近些年发展极为迅速．截至目前最重要的几种现代生成式模型包括 GAN、VAE、扩散模型、流模型和自回归模型等．其中，扩

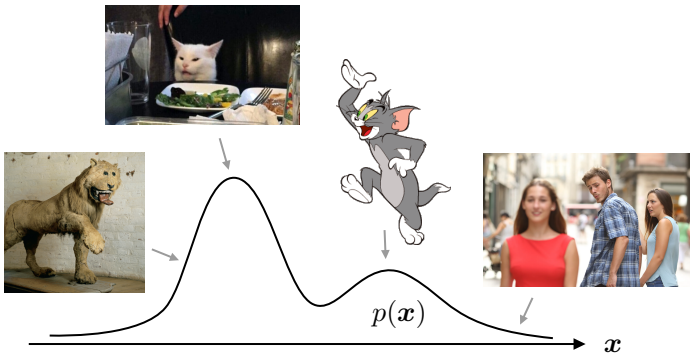
散模型是近些年最火的模型，它是许多有名的 AI 平台背后的算法，例如 Stable Diffusion 和 DALL·E 的核心就是扩散模型.

1.2　生成式模型的共同原理

图像生成的算法虽然很多，但是他们中很多都有着类似的思路. 下面我们就先来介绍一下它们的通用思路. 图像生成算法的基本任务是生成图像. 并且，我们希望模型能“懂” 一大类图像应该长什么样子，能真的生成新的、不一样的图像，而不是只会记住一些固定的图片. 也就是说，模型中必须带有一些随机性. 因此，我们需要用统计学的观点去思考图像生成的模型.

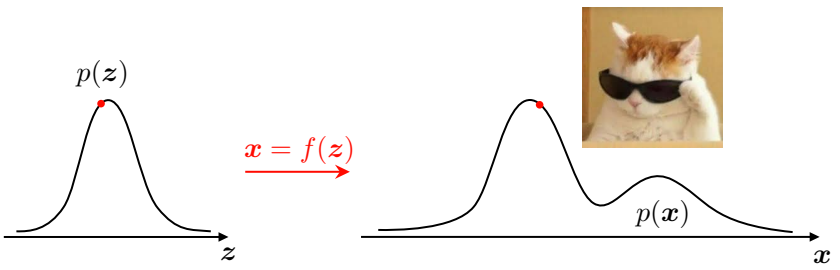
我们知道，在统计学看来，一张图像可以视作高维空间中的一个点，或者说一个向量. 例如一张 512×512 像素的 RGB 图片，它可以以 $512 \times 512 \times 3$ 个数来描述，也就可以看作这么高维空间中的一个点或向量. 我们后面将一直使用字母 $\boldsymbol{x}$ 表示我们想要生成的图像的向量. 而我们想要生成的所有可能的图片则在这个高维空间中形成了一个概率分布 $p(\boldsymbol{x})$. 举个例子，所有“猫” 的图片就构成了一个分布，其简化示意图如下图所示. 分布中概率密度越高的点，代表这个图像就越像是一只猫，而不像是猫的图片的概率则几乎为零. 生成式模型所需要做的，就是从这个分布中采样，即可以生成一张猫猫图片.

你可能会认为，这样采样的话会有一定概率采到不是猫的图片. 但实际上，由于我们采样的空间非常大，采到这些小概率图片几乎是不可能的.



可是，这个分布往往维数极高，并且过于复杂，我们既无法直接求出这个分布，也难以在这种分布中进行采样. 因此，我们需要找到一种间接的采样方法. 不同的生成式模型算法，其实就是以不同的思路去间接进行这个采样.

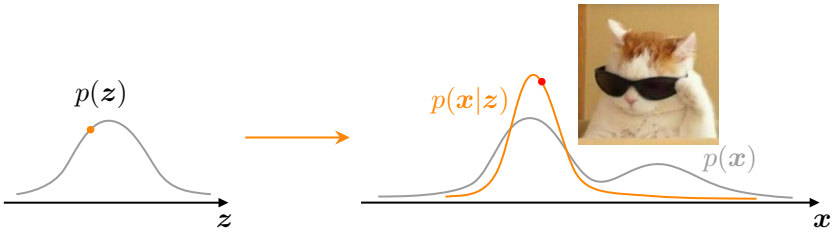
我们知道，计算机可以很容易地从一些简单的分布——如正态分布、均匀分布等——里采样. 很自然地，一些算法的思路就是找到一个函数 $\boldsymbol{x} = f(\boldsymbol{z})$，将一个分布简单的变量 $\boldsymbol{z}$ 转化为我们想要的分布复杂的变量. 当然，这个函数 f 往往会非常复杂，需要使用一个神经网络通过学习数据——也就是这个分布里已有的采样——来近似. GAN 和流模型就是这样的思路.



我们通常把这个分布简单的变量 $\boldsymbol{z}$ 称为 **隐变量** (latent variable). 隐变量可以被理解成对于输出变量 $\boldsymbol{x}$ 的一种描述，例如表示图片中猫的颜色、大小、品种、动作等. 当然，

真实算法中学到的隐变量不一定有这么直接而明确的意义，但我们一定程度上还是可以这样去理解它．因此，生成式模型可以看作模拟根据一个概念如“躺着的胖橘猫” 生成一个真实的猫的过程．

你可能会说，“躺着的胖橘猫” 也可以生成无数张图像啊．没错，在给定一个隐变量 $\boldsymbol{z}$ 后，我们也可以假定输出变量 $\boldsymbol{x}$ 不是一个确定的值，而是一个随机变量，满足某种条件分布 $p(\boldsymbol{x}|\boldsymbol{z})$．我们同样可以用一个神经网络去近似这个分布，在这个分布中进行采样，得到最终的输出．当然，既然要在 $p(\boldsymbol{x}|\boldsymbol{z})$ 这个分布里采样，那这个分布也需要比较简单，比如是高斯分布．这种思路被用于 VAE 和扩散模型中．



1.3　一些有用的资源

由于扩散模型乃至整个生成式模型的领域还相对比较新，每几年也都会有新的算法出现，因此目前并没有很好的教科书．好在 AI 领域是一个提倡公开、开源的领域，除了一些 AI 公司的最新模型属于商业机密以外，大部分常见的基本模型都是开源的．

例如 DALL·E 2 等模型目前并未公开具体细节．

一些教授、AI 工程师等人也喜欢写一些笔记或讲义．以下列举了笔者在学习过程中主要使用的资源．

1. Hugging Face：这是一个最常用的 AI 社区，其中即可以下载许多已经写好的 AI 模型，也可以作为论坛进行讨论；
2. 一些博客：许多人都喜欢在 GitHub 或个人主页上写一些 AI 模型的笔记或讲义，随便搜就能搜到．这里最推荐 Lilian Weng 的博文 和 苏剑林的博文；
3. arXiv：arXiv 上不仅有各个模型的原始论文，还有许多教授写的讲义，其中一些也很有帮助．

2　变分自编码器

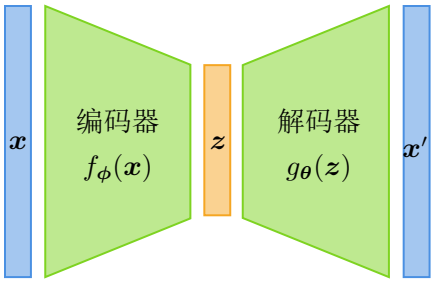
本章我们来介绍第一个隐变量模型——**变分自编码器** (Variational AutoEncoder, **VAE**). 从应用角度看，VAE 本身目前并不是任何有名的 AI 程序的主要算法，但它的理论价值却很高，也是扩散模型的基础.

2.1　传统自编码器

2.1.1　自编码器结构

在讨论变分自编码器前，我们先来看一看传统的自编码器. **自编码器** (autoencoder) 主要是一种降维算法，它使用一种特定结构的神经网络，其基本结构如下图所示.

自编码器的结构并不必须要对称，但为了简单方便起见，它经常被设计成对称的.



可以看到，自编码器是一个普通的前馈神经网络，它的输入层和输出层的宽度等于数据的原始维数，而中间有一个维度较低的“瓶颈”. 瓶颈的左侧称为 **编码器** (encoder)，右侧称为 **解码器** (decoder). 数据向量从输入层输入，经过编码器计算到中间的瓶颈层，再经过解码器，最终得到输出. 自编码器的训练目标是尽量准确地重构输入向量，即损失函数为

$$L = \frac{1}{n}\sum_{i=1}^{n} ||\boldsymbol{x}_i - \boldsymbol{x}_i'||^2$$

这样训练后，我们强迫编码器将数据里尽量多的信息编码在瓶颈处的低维编码 $\boldsymbol{z}$ 中，这样右侧的解码器才能从低维编码尽量准确地重构回高维数据. 因此，自编码器学习到的是一种对高维信息的有效低维编码，或者说一种隐变量.

2.1.2　自编码器作为生成模型

自编码器主要被用于降维，也可以用于数据降噪、异常点检测等. 但在这里，我们想要尝试一下自编码器的生成能力. 我们前面说过，生成模型就是通过隐变量生成新的数据，这个功能显然由自编码器中的解码器部分承担. 我们随机取一个隐变量，将它通过解码器，就可以生成新的数据了.

我们在这里训练了一个卷积自编码器，将 128×128 RGB 的猫猫图片降至 3072 维的隐变量，再重构回原始图片. 其表现如下图所示.

我们的数据集来自 AFHQ 数据集的猫猫图片，并被降至 128×128 像素.

可以看到，我们的自编码器大体上重构了输入的图像．然而如果我们要任意取一个隐变量用于生成新的图片，我们会遇到的问题是不知道该在哪里取．我们首先可以尝试一下在两个图片之间进行内插．也就是说我们将两个真实图片的隐变量之间等距地线性内插得到新的隐变量，再通过解码器生成图片．结果如下图所示．

可以看到，我们是可以得到一些有意义的图片的，不过这实际上也并没有生成全新的图片，而只是图片之间的内插．如果我们真的选取一些随机的隐变量来生成图片，则大概率会得到一些无意义的图片．例如下图中，我们甚至已经让选取的隐变量的各个分量的大致范围与真实图片的隐变量相近了，但仍然得到的是无意义的噪声．

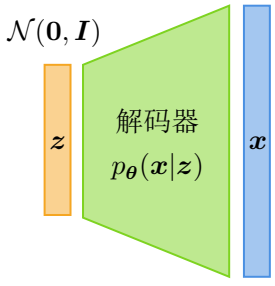
由此我们可以说明，传统自编码器是有一定生成能力的．但由于我们不知道隐变量的分布，因此难以真正随机地产生全新的图片．

2.2　VAE 的基本原理

2.2.1　从 AE 到 VAE

我们已经看到了，传统自编码器的生成能力有限，这是因为我们不知道隐变量的分布是什么样子的，自然也无法进行采样． VAE 为了解决这个问题，直接显性地对隐变量的分布和生成过程进行建模．我们假设假设隐变量 $\boldsymbol{z}$ 服从一个我们已知的先验分布—— 这个分布可以是我们任意选取的，为了方便我们就选择标准正态分布 $\mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$．隐变量通过某种我们未知的过程生成真实数据 $\boldsymbol{x}$，这个过程可以通过条件分布 $p(\boldsymbol{x}|\boldsymbol{z})$ 描述．这个真实的条件分布我们并不知道，因此我们需要通过对它使用神经网络进行估计，我们估计的条件分布记作 $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$．这个神经网络由 $\boldsymbol{z}$ 生成 $\boldsymbol{x}$，自然就是一个解码器．

所有不带参数的如 $p(\boldsymbol{x}|\boldsymbol{z})$ 都表示真实的分布，而所有带参数的如 $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$ 和后面的 $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ 等都表示对真实分布的一种参数化的估计，也就是一个神经网络．

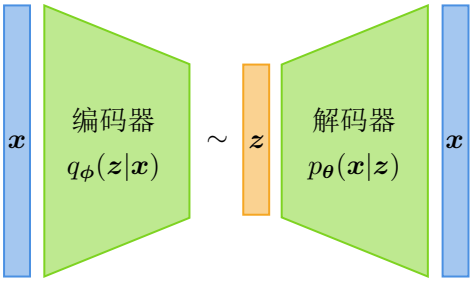


在这里，我们希望最大化真实数据出现的概率，即 $p_{\boldsymbol{\theta}}(\boldsymbol{x})$．从理论上我们有 $p_{\boldsymbol{\theta}}(\boldsymbol{x}) = \mathbb{E}_{\boldsymbol{z}\sim p(\boldsymbol{z})}[p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})]$．这样，我们可以对 $\boldsymbol{z}$ 进行大量采样并生成 $\boldsymbol{x}$，由此估计出 $p_{\boldsymbol{\theta}}(\boldsymbol{x})$． 但实际上，由于隐变量空间通常是很高维的，而对于任意一个给定的样本 $\boldsymbol{x}^{(i)}$，又只有很小的区域的 $\boldsymbol{z}$ 能够较好地重构它，因此这样在整个隐变量的先验分布上采样估计在实际上是不现实的．

我们希望生成式模型能生成大量不同的样本，所以不能让隐变量空间中的很大区域都会重构出同样的样本．

那么该怎么办呢？很自然地想到，对于给定的样本，我们只需要知道隐变量空间中哪些部分能够重构它，只从这个分布里采样就好了．这显然就是后验分布 $p(\boldsymbol{z}|\boldsymbol{x})$．当然，这个最真实的后验我们是无从得知的，因此我们只能近似我们估计的生成过程的后验 $p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})$，我们把它称为真实后验．然而，直接求解这个真实后验也是不行的，因为贝叶斯公式在分母中仍然是我们难以估计的 $p_{\boldsymbol{\theta}}(\boldsymbol{x})$．那么我们只好再用另一个神经网络 $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ 去估计这个后验．这个神经网络由 $\boldsymbol{x}$ 生成 $\boldsymbol{z}$，正好类似于传统自编码器的编码器部分．这样，我们的神经网络架构就成了下图的样子．

其实 $p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})$ 仍然是估计的生成过程 $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$ 而非真实的生成过程 $p(\boldsymbol{x}|\boldsymbol{z})$ 的后验，但在 VAE 里我们就叫它真实后验，因为后面我们再用 $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ 近似了这个“真实”的后验．



这样，整个模型的架构就很类似于自编码器了．这就是 VAE 的大体架构．这里要注意的是，编码器输出的是 $\boldsymbol{z}$ 的后验分布，解码器的输入则是其中的一个采样 $\boldsymbol{z}$．我们后面会讲解编码器网络该如何输出一个分布．

2.2.2　变分推断与 ELBO

VAE 既然是基于生成的概率模型的，它的训练目标自然就不再是最小化重构误差，而是最大化生成的概率 $p_{\boldsymbol{\theta}}(\boldsymbol{x})$，或者对数概率 $\log p_{\boldsymbol{\theta}}(\boldsymbol{x})$．我们说过，这个概率本身是难求的．不过，我们可以通过统计学里的 **变分推断** (variational inference) 的方法来找到一个合适的目标函数．我们将 $\log p_{\boldsymbol{\theta}}(\boldsymbol{x})$ 变一下形，可以得到

第一个等号是因为对 $\boldsymbol{z}$ 求期望不影响 $\boldsymbol{x}$ 的函数．

$$
\begin{aligned}
\log p_{\boldsymbol{\theta}}(\boldsymbol{x}) &= \mathbb{E}_{\boldsymbol{z}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x})\right] \\
&= \mathbb{E}_{\boldsymbol{z}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z})}{p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})}\right] \\
&= \mathbb{E}_{\boldsymbol{z}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z})}{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\right] + \underbrace{\mathbb{E}_{\boldsymbol{z}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}{p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})}\right]}_{D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})\,||\,p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}))}
\end{aligned}
\tag{2.1}
$$

其中最后一行的第二项即为 $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ 和 $p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})$ 的 KL 散度的定义，它一定不小于零．因此第一项一定不大于 $\log p_{\boldsymbol{\theta}}(\boldsymbol{x})$，我们称这一项为 **证据下界** (Evidence Lower Bound, **ELBO**)．即

两个分布 $p(x)$ 和 $q(x)$ 的 KL 散度 $D_{\mathrm{KL}}(p\,||\,q)$ 定义为

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

它代表 p 和 q 的一种“距离”．

$$
\mathrm{ELBO}(\boldsymbol{x},\boldsymbol{z}) := \mathbb{E}_{\boldsymbol{z}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z})}{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\right] \tag{2.2}
$$

ELBO 可以作为 $\log p_{\boldsymbol{\theta}}(\boldsymbol{x})$ 的一个下界，可以成为 VAE 的一个合适的目标函数．进一步看，由式 2.1 我们可以得到

$$
\mathrm{ELBO} = \log p_{\boldsymbol{\theta}}(\boldsymbol{x}) - D_{\mathrm{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})\,||\,p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})\right) \tag{2.3}
$$

最大化 ELBO，就相当于最大化 $\log p_{\boldsymbol{\theta}}(\boldsymbol{x})$ 的同时最小化 $D_{\mathrm{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})\,||\,p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})\right)$，也就是在最大化生成真实数据的概率的同时使得我们估计的后验 $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ 和真实后验 $p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})$ 尽量接近，而这正是我们引入 $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ 时想要的．也就是说最大化 ELBO 甚至比直接最大化 $\log p_{\boldsymbol{\theta}}(\boldsymbol{x})$ 还多了一些好处．

不过，ELBO 的定义仍然没法直接计算，因为它涉及了我们不知道的 $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$．我们可以将这一项拆开，进一步变形，得到

注意 $p(\boldsymbol{z})$ 是已知的先验，并不带参数 $\boldsymbol{\theta}$.

$$\begin{aligned}\mathrm{ELBO}(\boldsymbol{x},\boldsymbol{z}) &= \mathbb{E}_{\boldsymbol{z}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log\frac{p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z})}{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\right] \\ &= \mathbb{E}_{\boldsymbol{z}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log\frac{p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})}{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\right] \\ &= \mathbb{E}_{\boldsymbol{z}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right] - \underbrace{\mathbb{E}_{\boldsymbol{z}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log\frac{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}{p(\boldsymbol{z})}\right]}_{D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})\,||\,p(\boldsymbol{z}))}\end{aligned} \tag{2.4}$$

这个式子也有直观的意义．其中第一项是解码器的目标函数，它代表数据重构得有多好——对于给定的数据 $\boldsymbol{x}$，我们在编码器输出的分布 $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ 中采样 $\boldsymbol{z}$，并计算重构出解码器原始数据的概率 $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$ 的对数期望，这就是整个 VAE 重构出原始数据的对数期望．而第二项则是编码器的目标函数，也可以看作一种正则化——它使得我们估计的后验 $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ 更接近 $\boldsymbol{z}$ 的真实分布，这样方便我们从已知的 $\boldsymbol{z}$ 的分布中采样用于生成新的数据．

2.2.3　VAE 的具体架构与目标函数

我们在式 2.4 中得到了 VAE 要最大化的目标函数．这个函数涉及三个分布：$p(\boldsymbol{z})$、$p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$ 和 $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$．其中 $p(\boldsymbol{z})$ 我们已经在最开始假设了是一个标准正态分布，而后面的两个则是用神经网络去近似的两个分布．为了方便，我们可以进一步假设 $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$ 也是一个正态分布，只不过是一个很复杂的正态分布，我们需要用神经网络来近似．这样，它的真实后验 $p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})$ 也一定是一个正态分布． 从而我们可以让我们估计的后验 $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ 也是一个正态分布．要让神经网络定义出这两个正态分布，我们可以让神经网络输出它们的均值和方差，也就是

先验和似然都是正态分布的时候，后验也一定是正态分布.

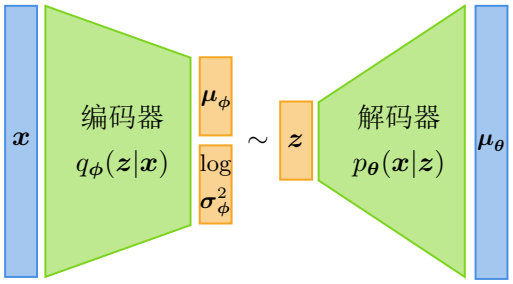
$$\begin{aligned}q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) &= \mathcal{N}\left(\boldsymbol{z}\,|\,\boldsymbol{\mu}_{\boldsymbol{\phi}}(\boldsymbol{x}),\mathrm{diag}(\boldsymbol{\sigma}^2_{\boldsymbol{\phi}}(\boldsymbol{x}))\right) \\ p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) &= \mathcal{N}\left(\boldsymbol{x}\,|\,\boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{z}),\mathrm{diag}(\boldsymbol{\sigma}^2_{\boldsymbol{\theta}}(\boldsymbol{z}))\right)\end{aligned}$$

对于这个式子，我们有两点需要指出的：

1. 注意到我们在这两个式子里又做了一个假设，也就是假设隐变量的各个维度是独立的．这个假设也是为了方便，因为这种情况下，它们的协方差矩阵一定是对角矩阵，因此我们只需要让神经网络输出一个“方差向量” 就可以了．实际上，为了数值稳定性的原因，我们一般输出的是对数方差 $\log\boldsymbol{\sigma}^2$；
2. 实际上在实现 VAE 时，我们经常设定 $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$ 的各个维度的方差都是 1，也就是其协方差矩阵为单位矩阵．这样，我们就不用输出它的方差了，直接输出均值就可以了．而均值实际上就是我们重构的输入 $\boldsymbol{x}$．

任何多维正态分布都可以通过一个线性变换将各个维度变为独立的，因此神经网络里的一个线性层就可以让各个维度之间变得独立.

经过了这些修改后，最终，我们的 VAE 的神经网络架构变成了下图．



将两个神经网络输出的分布定义成参数化的正态分布以后，我们就可以计算我们的目标函数，也就是 ELBO 了．为了与其他算法一致，我们在这里计算 VAE 的损失函数，也就是负的 ELBO．对于式 2.4 的第一项，也就是重构项 $\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$，我们可以将正态分布的均值和方差代入，并把期望变成采样均值，得到

$$-\mathbb{E}_{\boldsymbol{z}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right] = \frac{1}{2}\underset{\boldsymbol{z}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}{\operatorname{mean}}||\boldsymbol{x}-\boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{z})||^2 \tag{2.5}$$

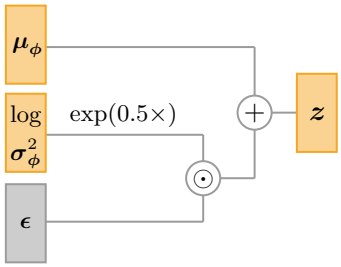
可以发现这就是我们所熟悉的均方误差损失 (MSE loss)，这和传统自编码器是一样的．注意这里的均值的意思是，对于每一个数据点 $\boldsymbol{x}^{(i)}$，在编码器给定了分布 $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}^{(i)})$ 后，我们在其中可以多次对 $\boldsymbol{z}$ 进行采样，进而使用解码器生成多个重构输出，用这些重构输出与给定的这一个 $\boldsymbol{x}^{(i)}$ 计算 MSE 误差．这个误差再对每个 $\boldsymbol{x}^{(i)}$ 求和，得到总重构误差．

而第二项的 KL 散度对于两个正态分布也是有闭式解的．对于给定的一个数据点 $\boldsymbol{x}^{(i)}$ 生成的后验 $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}^{(i)}) = \mathcal{N}(\boldsymbol{z}\,|\,\boldsymbol{\mu}_{\boldsymbol{\phi}}, \operatorname{diag}(\boldsymbol{\sigma}_{\boldsymbol{\phi}}^2))$，它和 $p(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{z}\,|\,\boldsymbol{0}, \boldsymbol{I})$ 的 KL 散度为

$$D_{\mathrm{KL}}\left(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})\,||\,p(\boldsymbol{z})\right) = \frac{1}{2}\left(-d + \boldsymbol{\mu}_{\boldsymbol{\phi}}^2 + \boldsymbol{\sigma}_{\boldsymbol{\phi}}^2 - \log\boldsymbol{\sigma}_{\boldsymbol{\phi}}^2\right) \tag{2.6}$$

其中 d 是隐变量 $\boldsymbol{z}$ 的维度．同样，这一项对每个 $\boldsymbol{x}^{(i)}$ 求和，就可以得到总的正则项．这样，将这两项相加，我们就得到了 VAE 的整个损失函数．

最后，我们需要指出的一点是，我们从 $\mathcal{N}(\boldsymbol{z}\,|\,\boldsymbol{\mu}_{\boldsymbol{\phi}}, \operatorname{diag}(\boldsymbol{\sigma}_{\boldsymbol{\phi}}^2))$ 里采样得到 $\boldsymbol{z}$ 的过程是随机的，所以不可微，梯度在这里无法回传．这个问题解决方式称为 **重参数化技巧** (reparameterization trick)，即把随机的采样变为确定的 $\boldsymbol{z} = \boldsymbol{\mu}_{\boldsymbol{\phi}} + \boldsymbol{\sigma}_{\boldsymbol{\phi}} \odot \boldsymbol{\epsilon}$，其中 $\boldsymbol{\epsilon}$ 是一个标准高斯噪声，$\odot$ 表示对应元素相乘．其计算图如下图所示．这样，梯度就可以回传到编码器里用于优化了．



3　两个基本算法——DDPM 与 NCSN

扩散模型 (diffusion models) 是近些年最火的生成式模型，甚至可以说没有之一．扩散模型的理论基础在 2010 年代就基本建立了，但真正成功地实现高质量图片生成是到了 2020 年左右被独立提出的两个模型——DDPM 和 NCSN．因此，这两个模型也被视作现代扩散模型的基础．我们这章就来具体地介绍这两个模型的统计学原理．

3.1　基于正向和反向扩散的 DDPM 算法

我们要介绍的第一种算法叫做 **DDPM** (Denoising Diffusion Probabilistic Models)，它由 Ho et al. 于 2020 年提出，立刻掀起了扩散模型的热潮，从而被认为是第一个成功生成高质量图片的扩散模型．

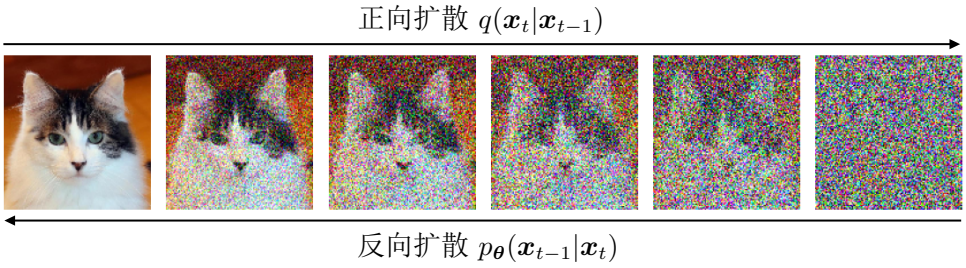
3.1.1　正向与反向扩散过程

DDPM 考虑的过程是所谓的扩散过程，也就是向真实数据中逐渐加入噪声的过程．对于一个数据点 $\boldsymbol{x}$，或者我们叫它 $\boldsymbol{x}_0$，我们可以向其中一点一点加入高斯噪声，构造出一个正向扩散序列 $\boldsymbol{x}_0 \to \boldsymbol{x}_1 \to \cdots \to \boldsymbol{x}_T$．具体来说，DDPM 以如下的方式加入噪声

$$\begin{aligned} \boldsymbol{x}_t &= \sqrt{\alpha_t}\boldsymbol{x}_{t-1} + \sqrt{1-\alpha_t}\boldsymbol{\epsilon}_{t-1} \\ \text{或写成}\quad q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}) &= \mathcal{N}(\boldsymbol{x}_t \,|\, \sqrt{\alpha_t}\boldsymbol{x}_{t-1}, (1-\alpha_t)\boldsymbol{I}) \end{aligned} \tag{3.1}$$

$1-\alpha_t$ 也被写作 β_t．

其中 $\boldsymbol{\epsilon}_t$ 是独立同分布的标准高斯噪声，而 α_t 是一个参数．这样定义扩散过程的原因是，当 $T \to \infty$ 时，可以证明 $\boldsymbol{x}_T$ 趋向于标准高斯噪声 $\mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$．



这样，我们可以把 $\boldsymbol{x}_T$ 当作一种隐变量，从标准高斯噪声中采样得到 $\boldsymbol{x}_T$，并由它逆回去构建出原始数据 $\boldsymbol{x}_0$．当然，从 $\boldsymbol{x}_T$ 一步估计出 $\boldsymbol{x}_0$ 是不现实的，而比较现实的做法是一点一点把噪声去掉，也就是做一个扩散过程的逆过程 $\boldsymbol{x}_T \to \boldsymbol{x}_{T-1} \to \cdots \to \boldsymbol{x}_0$．这需要我们求出反向扩散过程每一步的分布函数 $q(\boldsymbol{x}_t|\boldsymbol{x}_{t+1})$．和 VAE 类似，这个后验也难以直接求解，需要使用一个神经网络近似，我们把它记作 $p_{\boldsymbol{\theta}}(\boldsymbol{x}_t|\boldsymbol{x}_{t+1})$．估计 $p_{\boldsymbol{\theta}}(\boldsymbol{x}_t|\boldsymbol{x}_{t+1})$，就是扩散模型的核心．

扩散模型和 VAE 的符号使用不太一样，q 表示扩散过程，$p_{\boldsymbol{\theta}}$ 表示对后验的变分估计．它们的共同逻辑是，p 代表由隐变量生成数据的方向，q 代表由数据推断隐变量的方向．

在此我们还需要先推导一个后面会经常用到的关系式．对于整个扩散过程，若我们想要对中间的某步 $\boldsymbol{x}_t$ 进行采样，我们不需要一步一步向里面加入高斯噪声，而可以直接一步求出它的分布．使用 3.1 重复带入，可以很容易求出

$$\begin{aligned} \boldsymbol{x}_t &= \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\bar{\boldsymbol{\epsilon}}_{t-1} \\ \text{或写成}\quad q(\boldsymbol{x}_t|\boldsymbol{x}_0) &= \mathcal{N}(\boldsymbol{x}_t \,|\, \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0, (1-\bar{\alpha}_t)\boldsymbol{I}) \end{aligned} \tag{3.2}$$

DDPM 原始论文和大多数地方都把这里的噪声直接写作 $\boldsymbol{\epsilon}$，但为了后面的推导更加清晰没有歧义，我们选择了这种写法以显示出更多信息．

其中 $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$，而 $\bar{\boldsymbol{\epsilon}}_{t-1}$ 仍然是一个标准高斯噪声，但它综合了从 $\boldsymbol{x}_0$ 到 $\boldsymbol{x}_t$ 所加入的所有噪声．这样，我们就可以从 $\boldsymbol{x}_0$ 直接求出任何一步的分布了，这个性质在后面将会非常有用．

3.1.2 近似反向扩散过程

与 VAE 一样，DDPM 也无法直接最大化 $\log p_{\boldsymbol{\theta}}(\boldsymbol{x}_0)$，而需要借助变分推断，最大化 ELBO．DDPM 的 ELBO 与式 2.3 十分类似，它的具体形式为

$$\text{ELBO} = \log p_{\boldsymbol{\theta}}(\boldsymbol{x}_0) - D_{\text{KL}}(q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0) \,||\, p_{\boldsymbol{\theta}}(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)) \tag{3.3}$$

$\boldsymbol{x}_{1:T}$ 就是指 $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_T$.

可以看到，最大化 ELBO 就是最大化产生真实数据的概率 $p_{\boldsymbol{\theta}}(\boldsymbol{x}_0)$，同时最小化变分估计 $p_{\boldsymbol{\theta}}$ 与真实的反向扩散 q 过程在所有隐变量（包括扩散的中间步骤）$\boldsymbol{x}_{1:T}$ 上的差异．这是非常直观的．

不过，这个形式的 ELBO 仍然无法直接求．因此，我们需要将上式变一下形．经过一些复杂的数学推导，我们可以将 DDPM 的损失函数化为

$$\begin{aligned} L_{\text{DDPM}} &= -\text{ELBO} \qquad (3.4)\\ &= \mathbb{E}_{q(\boldsymbol{x}_{0:T})}\Bigg[\underbrace{-\log p_{\boldsymbol{\theta}}(\boldsymbol{x}_0|\boldsymbol{x}_1)}_{L_0} + \sum_{t=2}^{T} \underbrace{D_{\text{KL}}(q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0) \,||\, p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t))}_{L_{t-1}} + \underbrace{D_{\text{KL}}(q(\boldsymbol{x}_T|\boldsymbol{x}_0) \,||\, p(\boldsymbol{x}_T))}_{L_T}\Bigg] \end{aligned}$$

我们下面来分别解释一下这个损失函数中各项的意义

- L_0：这一项代表反向扩散的最后一步从 $\boldsymbol{x}_1$ 变为 $\boldsymbol{x}_0$ 有多好地重构了真实数据；
- L_{t-1}：这一项代表我们估计的反向扩散 $p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ 与真实后验 $q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0)$ 的相似程度；
- L_T：这一项代表先验的相似程度，它与模型参数 $\boldsymbol{\theta}$ 无关，所以优化时可以忽略．

下面我们就将这个损失函数化为可以直接优化的参数化形式．我们首先来处理这里面所有涉及正向扩散的部分．在忽略掉 L_T 这一项后，可以发现，唯一涉及到正向扩散的分布是 $q(\boldsymbol{x}_t|\boldsymbol{x}_{t+1}, \boldsymbol{x}_0)$，而这个分布是可以直接求解的．根据贝叶斯定理，我们有

$$q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0) = q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}, \boldsymbol{x}_0)\frac{q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_0)}{q(\boldsymbol{x}_t|\boldsymbol{x}_0)} = q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})\frac{q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_0)}{q(\boldsymbol{x}_t|\boldsymbol{x}_0)}$$

这是因为在给定 $\boldsymbol{x}_{t-1}$ 后，$\boldsymbol{x}_t$ 与 $\boldsymbol{x}_0$ 条件独立.

这里面的几个分布直接由 3.1 和 3.2 给出，且都是正态分布． 这样，$q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0)$ 也一定是一个正态分布，且它的协方差一定是各向同性的．它的均值和方差可以求出为

先验与似然都是正态时，后验一定也是正态．而后验是各向同性的是因为在扩散过程中我们一直增加的是各向同性的噪声，这样反向扩散显然也是各向同性的.

$$\begin{aligned} q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0) \text{ 的均值} \quad & \tilde{\boldsymbol{\mu}}_t(\boldsymbol{x}_t, \boldsymbol{x}_0) = \frac{(1-\bar{\alpha}_{t-1})\sqrt{\alpha_t}}{1-\bar{\alpha}_t}\boldsymbol{x}_t + \frac{(1-\alpha_t)\sqrt{\bar{\alpha}_{t-1}}}{1-\bar{\alpha}_t}\boldsymbol{x}_0 \\ \text{方差} \quad & \sigma_t^2 = \frac{(1-\bar{\alpha}_{t-1})(1-\alpha_t)}{1-\bar{\alpha}_t} \end{aligned}$$

下面我们再来处理 $p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ 这一项．为了方便，我们设定它的方差与 $q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0)$ 一样，都为 σ_t^2．而它的均值我们使用一个神经网络来估计，我们将这个神经网络记作 $\boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)$．也就是

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) = \mathcal{N}(\boldsymbol{x}_{t-1} \,|\, \underbrace{\boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)}_{\text{神经网络}}, \sigma_t^2\boldsymbol{I})$$

这样，我们的神经网络 $\boldsymbol{\mu}_{\boldsymbol{\theta}}(\cdot, t)$ 所实现的就是反向扩散过程——它的输入是带有噪声的图片 $\boldsymbol{x}_t$，而输出则是对去掉一步噪声的估计 $\boldsymbol{x}_{t-1}$．这就是 DDPM 名字中“去噪” 一词的来源．

有了两个条件分布的均值和方差，我们就可以直接求出单步损失 L_0 和 L_{t-1} 了．我们可以代入式 3.4，得到

两个方差相同的各向同性高斯分布 $\mathcal{N}(\boldsymbol{\mu}_1, \sigma^2\boldsymbol{I})$ 和 $\mathcal{N}(\boldsymbol{\mu}_2, \sigma^2\boldsymbol{I})$ 的 KL 散度正比于均值之间距离的平方，即为 $||\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2||^2/(2\sigma^2)$.

$$L_0 = \frac{1}{2\sigma_1^2}||\boldsymbol{x}_0 - \boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{x}_1, 1)||^2 + C \qquad L_{t-1} = \frac{1}{2\sigma_t^2}||\tilde{\boldsymbol{\mu}}_t(\boldsymbol{x}_t, \boldsymbol{x}_0) - \boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)||^2$$

其中我们将一些与模型参数 $\boldsymbol{\theta}$ 无关的项合并到了常数项 C 里．进一步可以发现，如果我们把 $\tilde{\boldsymbol{\mu}}_1(\boldsymbol{x}_1, \boldsymbol{x}_0)$ 定义为 $\boldsymbol{x}_0$ 的话，L_0 与其他项的形式是完全相同的．把它们综合在一起，我们就得到了 DDPM 的完整损失函数

$$L_{\text{DDPM}} = \sum_{t=1}^{T} \frac{1}{2\sigma_t^2} ||\tilde{\boldsymbol{\mu}}_t(\boldsymbol{x}_t, \boldsymbol{x}_0) - \boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)||^2 \tag{3.5}$$

这样，我们就有了完整的、可以直接进行优化的损失函数．可以看到，它具有我们熟悉的 MSE 损失的形式．我们只需用它来训练一个神经网络 $\boldsymbol{\mu}_{\boldsymbol{\theta}}(\cdot, t)$，用它将高斯噪声反向一步一步去噪，就可以得到生成的图像了．不过，DDPM 在真正实现的过程中还做了另外一些调整，我们下面再进行讨论．

3.1.3 从预测图像到预测噪声

上一节我们已经推导出了 DDPM 的最重要的思想，不过 DDPM 真正实现的方式和上面介绍的略有一点差异．DDPM 所做的最重要的一个改动就是使用神经网络去估计噪声，而非估计去噪后的图片．如果我们将上面 $q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0)$ 的均值 $\tilde{\boldsymbol{\mu}}_t(\boldsymbol{x}_t, \boldsymbol{x}_0)$ 根据 3.2 变一下形，化为 $\boldsymbol{x}_t$ 和噪声 $\boldsymbol{\epsilon}$ 的函数，可以得到

$$\tilde{\boldsymbol{\mu}}_t(\boldsymbol{x}_t, \bar{\boldsymbol{\epsilon}}_{t-1}) = \frac{1}{\sqrt{\alpha_t}}\boldsymbol{x}_t + \frac{1-\alpha_t}{\sqrt{\alpha_t(1-\bar{\alpha}_t)}}\bar{\boldsymbol{\epsilon}}_{t-1} \tag{3.6}$$

这里的记号有一点不严谨，我们是根据 $\boldsymbol{x}_0$、$\boldsymbol{x}_t$ 和 $\bar{\boldsymbol{\epsilon}}_{t-1}$ 的关系将它变形为 $\boldsymbol{x}_t$ 和噪声 $\bar{\boldsymbol{\epsilon}}_{t-1}$ 的函数，而不是把 $\bar{\boldsymbol{\epsilon}}_{t-1}$ 作为 $\boldsymbol{x}_0$ 传入这个函数．

其中 $\bar{\boldsymbol{\epsilon}}_{t-1}$ 一定程度上可以视作从 $\boldsymbol{x}_0$ 到 $\boldsymbol{x}_t$ 的总噪声，只不过大小被重新放缩成了标准高斯噪声． DDPM 算法在实现时实际上使用神经网络来估计 $\bar{\boldsymbol{\epsilon}}_{t-1}$，也就是使用下式来间接估计 $\boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)$

$$\boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t) = \frac{1}{\sqrt{\alpha_t}}\boldsymbol{x}_t - \frac{1-\alpha_t}{\sqrt{\alpha_t(1-\bar{\alpha}_t)}} \underbrace{\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)}_{\text{神经网络}} \tag{3.7}$$

注意，虽然我们已知噪声的分布就是标准高斯分布，但是我们这里用神经网络预测的是具体的这个噪声的样本．

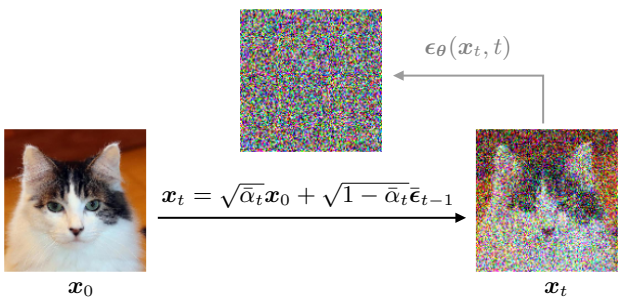
这样，我们的损失函数也需要跟着变化．可以求出，单步损失 L_t 仍然是 MSE 的形式，只不过变为正比于神经网络预测的噪声 $\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)$ 与真实添加的噪声 $\bar{\boldsymbol{\epsilon}}_{t-1}$ 的平方差异

$$L_t = \frac{1-\alpha_t}{2\alpha_t(1-\bar{\alpha}_t)} ||\bar{\boldsymbol{\epsilon}}_{t-1} - \boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)||^2$$

不过在 DDPM 的原始论文中，Ho et al. 发现如果把权重去掉，也就是把 L_t 直接简化成 MSE 损失，训练效果会更好．因此真正的 DDPM 直接使用 MSE 作为损失函数．

$$L_t^{\text{simple}} = ||\bar{\boldsymbol{\epsilon}}_{t-1} - \boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)||^2 \tag{3.8}$$

这样改进以后，还有另一点好处，那就是 L_t 只与 $\boldsymbol{x}_0$ 和 $\boldsymbol{x}_t$ 有关，而与 $\boldsymbol{x}_{t-1}$ 无关．因此我们无需生成整个正向扩散序列，而可以直接由 $\boldsymbol{x}_0$ 生成各个 $\boldsymbol{x}_t$，就可以优化这一步的 L_t．如下图所示．



总结起来，DDPM 的训练算法可以写作如下．我们每次随机选择一步 t，生成噪声并计算 $\boldsymbol{x}_t$，然后对模型参数进行一步更新．这样，对所有 t 重复后，就相当于对整个扩散序列进行了优化．

Algorithm 1 DDPM 训练

给定 $\{\alpha_t\}_{t=1}^T$、T

repeat

从数据集中采样 $\boldsymbol{x}_0$

随机抽取时间点 $t \in \{1, 2, \cdots, T\}$

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

计算 $\boldsymbol{x}_t = \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}$

将模型参数 $\boldsymbol{\theta}$ 以下式梯度进行更新

$$\nabla_{\boldsymbol{\theta}} ||\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)||^2$$

until 收敛

这里我们直接把噪声记作 $\boldsymbol{\epsilon}$ 而不是 $\bar{\boldsymbol{\epsilon}}_{t-1}$，因为它是直接一步采样得到的，而不是我们按照 3.1.3 小节里的逻辑从每次的噪声综合而来的．这也是为什么原文中直接把它记作了 $\boldsymbol{\epsilon}$.

而 DDPM 生成新数据的过程还是如我们最开始所说的，由纯噪声进行一步步的反向扩散，每次减掉一个小 $\boldsymbol{\epsilon}_{t-1}$，而不是试图直接一步减掉所有噪声得到 $\boldsymbol{x}_0$．其中每一步反向扩散由式 3.7 描述．

Algorithm 2 DDPM 生成

给定 $\{\alpha_t\}_{t=1}^T$、T

采样噪声 $\boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$

for $t = T, \cdots, 1$ **do**

if $t > 1$ **then** $\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$, **else** $\boldsymbol{z} = \boldsymbol{0}$

$$\boldsymbol{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)\right) + \sigma_t \boldsymbol{z}$$

end for

return $\boldsymbol{x}_0$

至此，我们就得到了 DDPM 的完整理论基础和算法．

3.1.4 DDIM 采样

DDPM 虽然取得了巨大的成功，但它也有很大的缺陷——它涉及非常多次（往往上千次）的反向扩散过程，才能从噪声生成有意义的图片，因此耗时非常长．为了解决这个问题，宋佳铭等人于 2021 年提出了一个改进策略，称为 **DDIM** (Denoising Diffusion Implicit Models)，可以在较短的步数内生成图片，极大的提高了生成速度．下面我们就来看一下 DDIM 的原理．

DDIM 首先试图找到一个包含了 DDPM 的更一般的框架，再在其中进行调整．观察 3.1.2 和 3.1.3 小节可以发现，DDPM 虽然在理论上是基于正向单步扩散 $q(\boldsymbol{x}_{t+1}|\boldsymbol{x}_t)$ 建立的，但在模型里实际上并没有用到这个分布． DDPM 模型真正使用到的只有两个分布：

1. 正向扩散的边际分布 $q(\boldsymbol{x}_t|\boldsymbol{x}_0)$，这个分布定义了我们直接使用的正向扩散过程，并用于推导反向扩散过程；
2. 真实反向扩散分布 $q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0)$，这个分布是用 $q(\boldsymbol{x}_{t+1}|\boldsymbol{x}_t)$ 推导出来的，它是反向扩散 $p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ 想要近似的分布．

反向扩散过程中 3.6 的推导使用了式 3.2.

这么看，边际分布 $q(\boldsymbol{x}_t|\boldsymbol{x}_0)$ 我们最好不要改，因为这样我们还可以使用 DDPM 模型的正向扩散过程．而 DDPM 中的反向扩散分布 $q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t,\boldsymbol{x}_0)$ 推导过程中用到了 $q(\boldsymbol{x}_{t+1}|\boldsymbol{x}_t)$．如果我们放弃对 $q(\boldsymbol{x}_{t+1}|\boldsymbol{x}_t)$ 的定义的话，我们可以直接定义反向扩散，只要它和 $q(\boldsymbol{x}_t|\boldsymbol{x}_0)$ 不矛盾就行．事实上，我们可以推导出满足这个条件的反向扩散分布

$$q_{\boldsymbol{\sigma}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t,\boldsymbol{x}_0)=\mathcal{N}\left(\boldsymbol{x}_{t-1}\middle|\sqrt{\bar{\alpha}_{t-1}}\boldsymbol{x}_0+\frac{\sqrt{1-\bar{\alpha}_{t-1}+\sigma_t^2}}{\sqrt{1-\bar{\alpha}_t}}(\boldsymbol{x}_t-\sqrt{\bar{\alpha}_t}\boldsymbol{x}_0),\sigma_t^2\boldsymbol{I}\right) \tag{3.9}$$

注意，这个 σ_t 和上面 DDPM 的 σ_t 不是一回事，DDPM 的 σ_t 是推导出来的，不是自由参数．另外，DDIM 原文将这里的 $\bar{\alpha}_t$ 记为了 α_t，但我们这里保持和上文记号一致．

其中这里有一族自由参数 σ_t，它决定了分布的方差，也对均值有影响．这样，我们的正向扩散的边际分布 $q(\boldsymbol{x}_t|\boldsymbol{x}_0)$ 仍然保持不变，但区别是现在这个扩散过程不再是马尔可夫性 (Markovian) 的了，也就是说 $\boldsymbol{x}_{t+1}$ 不再只由 $\boldsymbol{x}_t$ 决定，还和前面的中间过程有关．不过这对我们并没有什么影响，反正我们只需要使用 $q(\boldsymbol{x}_t|\boldsymbol{x}_0)$ 由 $\boldsymbol{x}_0$ 生成每个 $\boldsymbol{x}_t$．

由我们定义的 $q_{\boldsymbol{\sigma}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t,\boldsymbol{x}_0)$，我们可以和 DDPM 一样，使用神经网络去估计噪声，从而进行反向扩散．回顾 DDPM 的推导可以发现，这一部分是基于 $q(\boldsymbol{x}_t|\boldsymbol{x}_0)$ 的，而由于我们并没有改变这个边际分布，因此我们完全不需要进行任何改动，甚至可以直接把 DDPM 训练好的神经网络拿来用．

有了训练好的神经网络 $\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t,t)$，接下来我们就可以进行反向扩散了．这也是 DDIM 和 DDPM 真正体现出区别的地方．我们可以推导出在 3.9 定义的反向扩散分布下，反向扩散通过下式进行．

$$\boldsymbol{x}_{t-1}=\sqrt{\bar{\alpha}_{t-1}}\underbrace{\left(\frac{\boldsymbol{x}_t-\sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t,t)}{\sqrt{\bar{\alpha}_t}}\right)}_{\text{估计的 }\boldsymbol{x}_0}+\underbrace{\sqrt{1-\bar{\alpha}_{t-1}-\sigma_t^2}\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t,t)}_{\text{指向 }\boldsymbol{x}_t}+\sigma_t\boldsymbol{\epsilon}_t \tag{3.10}$$

此式对应着 DDPM 的 3.7 式.

可以看到，这里的噪声完全由自由参数 σ_t 决定．若我们令 σ_t 等于我们在 3.1.2 推出来的 σ_t，就得到了 DDPM．而若我们令 $\sigma_t=0$，那么反向扩散就完全变成了一个确定的过程，这种情况我们称作 DDIM．在这种情况下，既然反向扩散是完全确定的了，那我们理论上可以直接从 $\boldsymbol{x}_T$ 采样 $\boldsymbol{x}_0$．当然，我们的神经网络估计的总归是会有一些偏差的，所以实际使用中我们还是要一步步反向扩散的，但这回我们可以每步走的大一点．也就是说，我们可以选择 $0,\cdots,T$ 的一个子序列 $\tau_0,\cdots,\tau_S$，其中 $\tau_0=0,\tau_S=T$，在这个子序列中进行反向扩散

$$\boldsymbol{x}_{\tau_{i-1}}=\sqrt{\bar{\alpha}_{\tau_{i-1}}}\left(\frac{\boldsymbol{x}_{\tau_i}-\sqrt{1-\bar{\alpha}_{\tau_i}}\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_{\tau_i},\tau_i)}{\sqrt{\bar{\alpha}_{\tau_i}}}\right)+\sqrt{1-\bar{\alpha}_{\tau_{i-1}}}\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_{\tau_i},\tau_i) \tag{3.11}$$

我们在此式中已经令 $\sigma_t=0$.

实验发现，从过这种方式，我们可以将 1000 步的 DDPM 缩减到 20-100 步，而得到相似质量的图片．这就是 DDIM 速度更快的原理．当然，我们也可以将 σ_t 设置在中间的某个值，这样模型就介于 DDPM 和 DDIM 之间，具有一定的不确定性，但是也可以跳步取样，只不过跳的步子小一些．也就是说，σ_t 决定了模型在灵活性和速度之间的权衡．

3.2 基于分数的 NCSN 算法

我们将要介绍的第二种扩散模型算法称为 **NCSN** (Noise Conditional Score Network)，又称为 **SMLD** (Score Matching with Langevin Dynamics)，它是基于分数的方法 (score-based methods) 的代表． NCSN 由宋飏等人于 2019 提出，虽然早于 DDPM，但由于当时生成图像质量并没有很高，所以并没有称为第一个带起热度的扩散模型．但后来人们发现这个理论框架是对扩散模型的更高层次的理解，其理论价值甚至高于 DDPM．

3.2.1　Langevin 过程

我们在 1.2.1 小节中说过，图像生成任务本质上就是估计图像构成的分布，并从中采样的过程．然而，估计分布和采样这两个过程都非常复杂，因此人们发明了各式各样的生成式模型，来间接地实现这两个目标．

不过，现在假如我们通过某种方式知道了数据的真实分布 $p(\boldsymbol{x})$，那么是否有一种统计学方式，让我们可以在其中进行采样呢？答案是肯定的． 其中一种重要的采样算法称为无调整的 Langevin 算法 (unadjusted Langevin algorithm, ULA)．它起始于任意变量 $\boldsymbol{x}_0$，并以下式更新变量

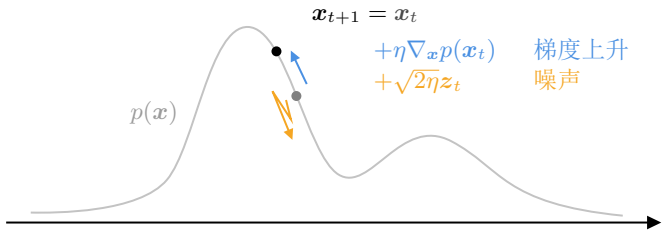
Langevin 动力学来自统计物理学里一种描述粒子分布的模型，它用于采样中可以被视作一种 MCMC 算法．

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + \eta \nabla_{\boldsymbol{x}} \log p(\boldsymbol{x}_t) + \sqrt{2\eta}\boldsymbol{z}_t \tag{3.12}$$

其中 $\boldsymbol{z}_t$ 是高斯噪声，η 是更新的步长． 可以证明，几乎对任意的分布 $p(\boldsymbol{x})$，当步长 η 足够小、且 $t \to \infty$ 时，$\boldsymbol{x}_t$ 的极限分布收敛于 $p(\boldsymbol{x})$．这样，在更新了很多步以后，$\boldsymbol{x}_t$ 就是对 $p(\boldsymbol{x})$ 的一个采样了．

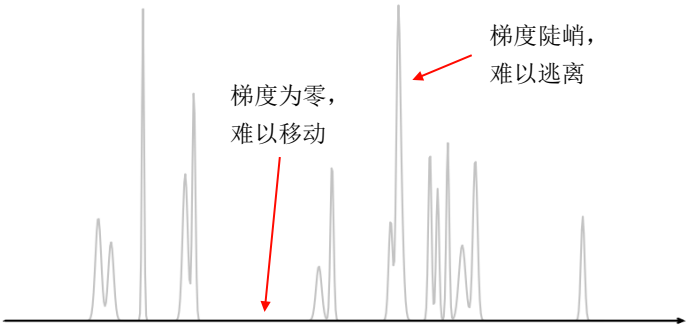
对于非无穷小的步长 η 和有限的时间 t，理论上需要所谓的 Metropolis-Hasting 修正，但在这里无需修正其性能已经足够好了．而分布“性质足够好”是说这个系统是遍历的 (ergodic)，在理论上任何连续分布几乎都满足遍历性条件．

下图显示了 Langevin 方程采样的原理．对于一个点 $\boldsymbol{x}_t$，$\nabla_{\boldsymbol{x}} p(\boldsymbol{x}_t)$ 指向 $p(\boldsymbol{x})$ 上升的方向，所以这一项会使得 $\boldsymbol{x}_{t+1}$ 倾向于取到概率更大的点．而噪声项 $\boldsymbol{z}_t$ 则增加了随机性，使得 $\boldsymbol{x}_{t+1}$ 有一定概率取到其他点，甚至有可能翻过局部极小值，进入另一个局部极大附近的区域．这样，当 t 足够大时，$\boldsymbol{x}_t$ 的分布就会与几乎符合 $p(\boldsymbol{x})$，这就达到了我们对 $p(\boldsymbol{x})$ 采样的目的．



不过真实的图像分布要比上图复杂得多，此时直接使用 Langevin 过程也会导致很大的问题．这是由于，在现实使用中，我们的采样步总是有限的，而且不能特别大．首先，在所有可能的图像中，只有极少部分的图像是真正有意义的． 也就是说，在整个图像空间中，绝大部分区域的概率密度几乎为 0．在这部分区域里，梯度也几乎为零向量，因此 Langevin 过程难以在有限步内将采样点移动到高概率的区域．另一方面，由于概率较高的区域往往是一个个分离的比较“瘦高” 的区域，称为 **模式** (mode)．模式区域里的梯度一般非常大，使得采样点难以在有限步内翻过低概率的区域进入其他的高概率区域．这会导致模型可能总是会采样出相似的输出，这种现象在生成式模型中称为 **模式崩溃** (mode collapse)．我们后面会看到 NCSN 如何处理这两个问题．

在流形学习中，我们称所有可能的数据所在的最大的空间为**外围空间** (ambient space)，而绝大多数数据往往只位于它的一个很低维的子空间上，称为**支撑空间** (support space)．



3.2.2 分数匹配

在上述的 Langevin 过程中，我们并不直接需要数据的真实分布 $p(\boldsymbol{x})$，而只需要它的梯度 $\nabla_{\boldsymbol{x}} p(\boldsymbol{x})$．我们把这个梯度称为 **Stein 分数** (Stein's score)，在扩散模型的领域里直接简称为分数．所谓基于分数的算法，就是使用这个分数进行 Langevin 采样的算法．不过，既然我们不知道 $p(\boldsymbol{x})$，自然也就无法直接求出它的梯度，而只能通过其他方式间接估计．这种估计分数的过程就称为 **分数匹配** (score matching)．

我们前面已经看到，在深度学习领域中，对于难以计算的量，我们都可以使用神经网络来估计．在这里也是如此，我们可以用一个神经网络 $\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x})$ 来近似 $\nabla_{\boldsymbol{x}} p(\boldsymbol{x})$．即最小化

$$\mathbb{E}_{\boldsymbol{x}\sim p(\boldsymbol{x})}\big[\,\|\underbrace{\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x})}_{\text{神经网络}}-\nabla_{\boldsymbol{x}}\log p(\boldsymbol{x})\|^2\,\big]$$

当然，这个损失是没法直接使用的，因为我们不知道 $\nabla_{\boldsymbol{x}} p(\boldsymbol{x})$．因此，人们开发了几种分数匹配的方法．第一种称为 **分层分数匹配** (sliced score matching)，它使用一个标准正态分布的随机向量 $\boldsymbol{v}$ 进行投影，可以证明它最小化的损失函数为

$$\mathbb{E}_{p_{\boldsymbol{v}}\sim\boldsymbol{v}}\mathbb{E}_{\boldsymbol{x}\sim p(\boldsymbol{x})}\left[\boldsymbol{v}^{\top}\nabla_{\boldsymbol{x}}\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x})\boldsymbol{v}+\frac{1}{2}\|\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x})\|^2\right] \tag{3.13}$$

而第二种常见的分数匹配方法称为 **去噪分数匹配** (denoising score matching)，它的思路和我们下一小节要介绍的 NCSN 非常类似，所以我们来具体看一下．对于一个分布 $p(\boldsymbol{x})$，如果我们给它加上一个很小的噪声，即 $\tilde{\boldsymbol{x}}=\boldsymbol{x}+\boldsymbol{\epsilon}$，这可以视作把分布里的每一个点 $\boldsymbol{x}$ 变为一个分布 $q_\sigma(\tilde{\boldsymbol{x}}|\boldsymbol{x})$．这样添加噪声后，整个分布变为 $q_\sigma(\tilde{\boldsymbol{x}})=\int q_\sigma(\tilde{\boldsymbol{x}}|\boldsymbol{x})p(\boldsymbol{x})\mathrm{d}\boldsymbol{x}$．若噪声足够小，那么添加噪声后分布改变很小，我们可以用 $q_\sigma(\tilde{\boldsymbol{x}})$ 近似作为 $p(\boldsymbol{x})$ 的梯度．那么我们可以证明，估计 $q_\sigma(\tilde{\boldsymbol{x}})$ 的梯度等价于最小化下式

$$\mathbb{E}_{\tilde{\boldsymbol{x}},\boldsymbol{x}\sim q_\sigma(\tilde{\boldsymbol{x}}|\boldsymbol{x})p(\boldsymbol{x})}\left[\|\boldsymbol{s}_{\boldsymbol{\theta}}(\tilde{\boldsymbol{x}})-\nabla_{\tilde{\boldsymbol{x}}}\log q_\sigma(\tilde{\boldsymbol{x}}|\boldsymbol{x})\|^2\right]$$

而 $\boldsymbol{\epsilon}$ 可以是任意一个很小的噪声．我们可以将它定义为高斯分布 $\boldsymbol{\epsilon}\sim\mathcal{N}(\boldsymbol{0},\sigma^2\boldsymbol{I})$，这样有 $q_\sigma(\tilde{\boldsymbol{x}}|\boldsymbol{x})=\mathcal{N}(\tilde{\boldsymbol{x}}|\boldsymbol{x},\sigma^2\boldsymbol{I})$．这个分布可以直接求导，进而很容易得到最终的损失函数为

$$\mathbb{E}_{\tilde{\boldsymbol{x}},\boldsymbol{x}\sim q_\sigma(\tilde{\boldsymbol{x}}|\boldsymbol{x})p(\boldsymbol{x})}\left[\left\|\boldsymbol{s}_{\boldsymbol{\theta}}(\tilde{\boldsymbol{x}})-\frac{\tilde{\boldsymbol{x}}-\boldsymbol{x}}{\sigma^2}\right\|^2\right] \tag{3.14}$$

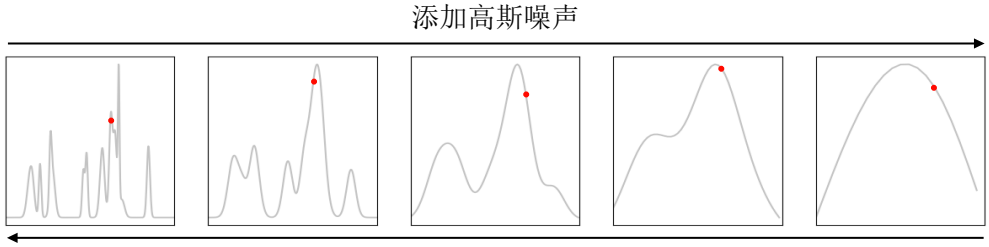
分层分数匹配和去噪分数匹配各有优劣——分层分数匹配拟合的是原始分布的分数，但时间复杂度更高；而去噪分数匹配更快，但它拟合的是添加了噪声的分布的分数，只有在噪声很小的时候才能代表原始分布的分数．

3.2.3 NCSN 与退火 Langevin 采样

现在我们已经有了估计分数的方式，最大的问题就是在 3.2.1 里讨论的采样困难．不过，去噪分数匹配给了我们一些灵感——如果给数据添加一些噪声，让数据的分布更加平缓，这个问题不就解决了吗？当然，我们增加的噪声需要足够大，才能完全解决梯度过缓或过陡的问题．但这样一来，数据的分布就会被改变太多了，采样也就不准确了．因此，我们面临一个权衡噪声大小的困境．

为了解决这个问题，人们提出了一个改进版的采样方案，称为 **退火 Langevin 动力学** (annealed Langevin dynamics, ALD)．退火 Langevin 动力学给数据分布添加了从小到大不同强度的噪声．采样时，我们先在噪声最大的分布里采样，然后使用 Langevin 方程

进行更新．但更新的过程中，我们会逐渐减小噪声．这样，在采样的早期，由于噪声较大，分布充满了整个空间，因此我们从任意地方开始都不会得到过小的梯度，从而可以有效地更新采样点．同时，由于各个模式内部的梯度也不会过于陡峭，因此采样点可以在不同模式之间随机跳跃，保证了最终采样有足够的随机性．随着噪声的减小，分布逐渐接近数据的真实分布．我们最终的采样便会符合我们想要的真实分布．



退火 Langevin 过程只是一个采样过程，它需要我们已知数据的分布的分数．现在我们假设我们已经训练好了一个神经网络 $\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x},\sigma)$．这个网络给出在噪声的强度为 σ 时，分布在 $\boldsymbol{x}$ 处的梯度，即 $\nabla_{\boldsymbol{x}} q_\sigma(\boldsymbol{x})$．这样，退火 Langevin 采样的伪代码为

Algorithm 3 退火 Langevin 采样

给定 $\{\sigma_i\}_{i=1}^L$、r、T，其中 $\sigma_L \approx 0$
随机初始化 $\boldsymbol{x}_0$
for $i = 1, \cdots, L$ **do**
　步长 $\eta_i = r \cdot \sigma_i^2 / \sigma_L^2$
　for $t = 1 \cdots T$ **do**
　　采样 $\boldsymbol{z}_t \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$
　　$\boldsymbol{x}_t = \boldsymbol{x}_{t-1} + \eta_i \boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}, \sigma_i) + \sqrt{\eta_i}\boldsymbol{z}_t$
　end for
　$\boldsymbol{x}_0 = \boldsymbol{x}_T$
end for
return $\boldsymbol{x}_0$

这里步长的公式基本上是通过经验得到的．

现在我们已经设计好了一个采样过程，只需要让神经网络 $\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x},\sigma)$ 去估计噪声扰动的分布 $q_\sigma(\boldsymbol{x})$ 的梯度，整个模型就完成了．这个神经网络 $\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x},\sigma)$ 就称为 NCSN 网络．NCSN 网络可以使用分层分数匹配或去噪分数匹配，但既然我们已经在给数据增加噪声了，那使用去噪分数匹配会更方便些．下面显示了以去噪分数匹配训练 NCSN 的伪代码．

Algorithm 4 NCSN 训练

给定 $\{\sigma_i\}_{i=1}^L$
repeat
　从数据集中采样 $\boldsymbol{x}$
　随机抽取噪声强度 $\sigma \in \{\sigma_1, \cdots, \sigma_L\}$，噪声 $\boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$
　为数据添加噪声 $\tilde{\boldsymbol{x}} = \boldsymbol{x} + \sigma\boldsymbol{\epsilon}$
　将模型参数 $\boldsymbol{\theta}$ 以下式梯度进行更新

$$\nabla_{\boldsymbol{\theta}} \left\| \boldsymbol{s}_{\boldsymbol{\theta}}(\tilde{\boldsymbol{x}}) - \frac{\tilde{\boldsymbol{x}} - \boldsymbol{x}}{\sigma^2} \right\|^2$$

until 收敛

以数据集先训练 NCSN 网络，再进行退火 Langevin 采样，这就是 NCSN 模型的完整过程.

3.2.4 NCSN 与 DDPM 的关系

我们看到，NCSN 涉及了向数据中加入从小到大不同强度的噪声，这点与 DDPM 十分类似. 那么二者是否有什么联系呢？答案是肯定的. 现在我们考虑 DDPM 的噪声调度，即以式 3.2 的方式向数据加入噪声. 那么此时的分数 $\nabla_{\boldsymbol{x}_t} \log q(\boldsymbol{x}_t)$ 可以化为

$$\begin{aligned}
\nabla_{\boldsymbol{x}_t} \log q(\boldsymbol{x}_t) &= \mathbb{E}_{q(\boldsymbol{x}_0)} \left[\nabla_{\boldsymbol{x}_t} \log q(\boldsymbol{x}_t | \boldsymbol{x}_0) \right] \\
&= \mathbb{E}_{q(\boldsymbol{x}_0)} \left[-\frac{\boldsymbol{x}_t - \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0}{1 - \bar{\alpha}_t} \right] \\
&= \mathbb{E}_{q(\boldsymbol{x}_0)} \left[-\frac{\bar{\boldsymbol{\epsilon}}_{t-1}}{\sqrt{1 - \bar{\alpha}_t}} \right] \\
&= -\frac{\bar{\boldsymbol{\epsilon}}_{t-1}}{\sqrt{1 - \bar{\alpha}_t}}
\end{aligned} \tag{3.15}$$

也就是说，分数其实就是噪声的反方向. DDPM 使用神经网络 $\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)$，和 NCSN 使用神经网络 $\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x}, \sigma)$，它们估计的基本上是同一个东西的相反两个方向. 而在 DDPM 中的采样中，我们是向噪声的反方向移动；在 NCSN 中，我们向分数的方向移动. 这本质上是一样的.

4　从微分方程的视角看扩散模型

上一节最后 DDPM 和 NCSN 的统一让我们猜测，应该有一个更高观点的理论框架可以统一各种扩散模型． 2021 年，NCSN 的发明者宋飏又发表了一篇著名的论文 Score-based generative modeling through stochastic differential equations，将 DDPM 与 NCSN 统一到了同一个数学模型的框架下——即 **随机微分方程** (stochastic differential equation, **SDE**)．这篇文章开启了扩散模型研究的新篇章．我们下面就跟着这篇文章的思路，从 SDE 的高观点重新理解一遍扩散模型．

4.1　随机微分方程

4.1.1　SDE 简介

我们首先来学习一下随机分析与随机微分方程的基本概念．熟悉这部分的读者可以直接跳过本小节．

考虑一个随时间 t 连续变化的一维随机变量 $X(t)$，我们称之为一个 **随机过程** (stochastic process)．关于 $X(t)$ 的一个随机微分方程是指形如下式的方程

为了方便，这里的时间 t 经常被写作下角标，例如 $X(t)$ 记作 X_t，$\mu(X,t)$ 记作 $\mu_t(X)$ 等．

$$\mathrm{d}X(t) = \mu(X,t)\mathrm{d}t + \sigma(X,t)\mathrm{d}W \tag{4.1}$$

其中字母 d 代表微分，它和我们在微积分里学习的微分的概念基本一样．$\mathrm{d}X(t)$ 表示在无限短的时间 $\mathrm{d}t$ 里 $X(t)$ 的变化量．根据这个方程，它由两部分决定：

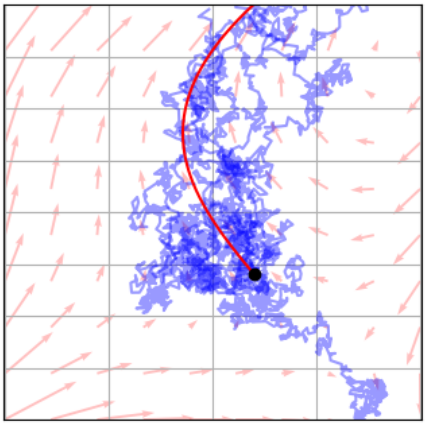
1. $\mu(X,t)\mathrm{d}t$，这项称为 **漂移** (drift) 项，它代表 $X(t)$ 的“定向移动” 的速度；
2. $\sigma(X,t)\mathrm{d}W$，这项称为 **扩散** (diffusion) 项，它为 $X(t)$ 的变化增加了随机性． 其中 $\mathrm{d}W$ 可以视作一个无穷小的高斯噪声，它服从 $\mathcal{N}(0,\mathrm{d}t)$．

$\mathrm{d}W$ 也记作 $\mathrm{d}B$，它是 Wiener 过程（又称为 Brown 运动）的微分．

当然，在机器学习领域，我们关注的往往是高维的数据．而对于一个多维随机变量 $\boldsymbol{X}(t)$，它的 SDE 的一般形式被推广为

$$\mathrm{d}\boldsymbol{X}(t) = \boldsymbol{\mu}(\boldsymbol{X},t)\mathrm{d}t + \boldsymbol{\Sigma}(\boldsymbol{X},t)\mathrm{d}\boldsymbol{W} \tag{4.2}$$

下图显示了一个二维 SDE 的解的采样．为了方便作图，在这里我们将 SDE 被简化为 $\mathrm{d}\boldsymbol{X} = \boldsymbol{\mu}(\boldsymbol{X})\mathrm{d}t + \sigma\mathrm{d}\boldsymbol{W}$．也就是说，漂移项是一个随时间不变的向量场 $\boldsymbol{\mu}(\boldsymbol{X})$，在图中表示为浅红色箭头．而噪声在这里始终是一个固定的各向同性高斯噪声．图中用蓝色画出了从同一个初始位置出发的多次不同采样，而红色则表示噪声为零，也就是只有漂移项的解．可以看到，SDE 的解大体上随着漂移项的方向移动，但同时又会有随机的“扩散”，导致路径偏离只有漂移的红色路径．



对于最一般的 SDE 形式如式 4.2，它的漂移项 $\boldsymbol{\mu}(\boldsymbol{X},t)$ 也是时间 t 的函数，就相当于图中的向量场也随时间变化；而噪声 $\boldsymbol{\Sigma}(\boldsymbol{X},t)$ 是随时间和位置的矩阵函数，代表噪声可以随着时间、位置变化，而且不一定是各向同性的．但它的解仍然代表带有随机扩散的漂移过程．

最后，我们需要指出 SDE 和我们前面常见到的带噪声的递推方程的关系．式 4.2 的 SDE 可以视作如下的递推方程在 $\Delta t \to 0$ 下的连续化，其中 $\boldsymbol{\epsilon}$ 是一个标准高斯噪声．

$$\boldsymbol{X}(t+\Delta t)-\boldsymbol{X}(t)=\boldsymbol{\mu}(\boldsymbol{X},t)\Delta t+\boldsymbol{\Sigma}(\boldsymbol{X},t)\sqrt{\Delta t}\boldsymbol{\epsilon} \tag{4.3}$$

4.1.2　作为 SDE 的扩散模型

在扩散模型的领域中，样本 $\boldsymbol{x}$ 的正向扩散过程也可以视作满足 SDE 演化，这个 SDE 取决于我们对模型的设定．我们一般设定噪声与图像本身无关，也就是扩散系数与 $\boldsymbol{x}$ 无关，并且是各向同性的．此时，正向扩散的 SDE 可以写作

我们在这里和上一小节相比改变了一些字母记号，这是为了和扩散模型的文献一致．

$$\mathrm{d}\boldsymbol{x}=\boldsymbol{f}(\boldsymbol{x},t)\mathrm{d}t+g(t)\mathrm{d}\boldsymbol{W} \tag{4.4}$$

按照扩散模型的惯例，这里的 t 从连续 0 变为 T，其中 $t=0$ 时 $\boldsymbol{x}$ 服从我们想要的图像的分布，而 $t=T$ 时 $\boldsymbol{x}$ 服从简单的正态分布．式中的扩散项 $g(t)\mathrm{d}\boldsymbol{W}$ 就是我们向样本中加入的噪声，而漂移系数 $\boldsymbol{f}(\boldsymbol{x},t)$ 则表示我们对样本本身的处理，例如 DDPM 中将样本本身缩小一个系数．我们后面在 4.1.3 中会具体解释．

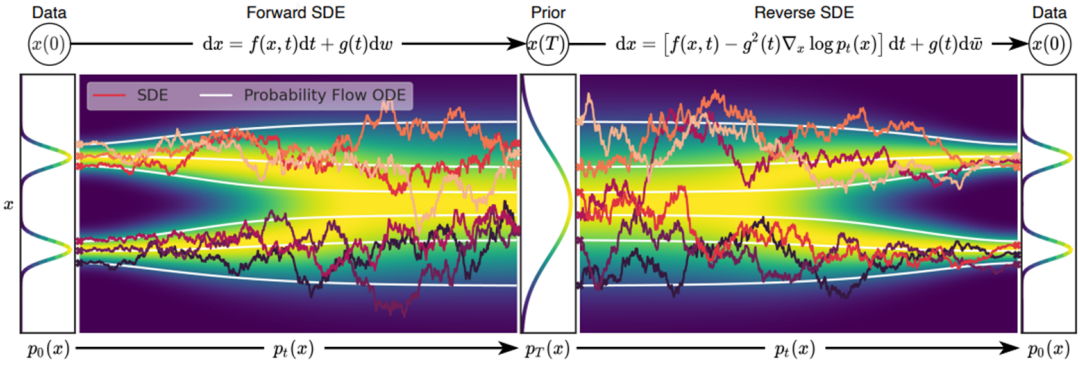
而扩散过程都是借助反向扩散过程进行采样的．每个正向扩散的 SDE 都对应于一个反向扩散的 SDE．可以证明，上式的反向扩散 SDE 为

$$\mathrm{d}\boldsymbol{x}=[\boldsymbol{f}(\boldsymbol{x},t)-g(t)^2\nabla_{\boldsymbol{x}}\log p_t(\boldsymbol{x})]\mathrm{d}t+g(t)\mathrm{d}\overline{\boldsymbol{W}} \tag{4.5}$$

其中 $p_t(\boldsymbol{x})$ 就代表时间为 t 时 $\boldsymbol{x}$ 的分布．这个逆向 SDE 有几点需要强调一下．首先，为了和正向扩散 SDE 中的 t 保持一致，这里的 t 是从 T 变为 0 的．第二，由于 SDE 的随机性，我们是不可能把具体每个采样轨迹给逆回去的．这里的逆向扩散是指分布保持一致．也就是说，如果我们从某个分布 $p_0(\boldsymbol{x})$ 开始，随正向扩散 SDE 演化，我们会得到每个时间的分布 $p_t(\boldsymbol{x})$．然后我们从最终时间的的分布 $p_T(\boldsymbol{x})$ 开始，再随逆向 SDE 演化，那么反向扩散得到每个时间的分布恰好和之前正向扩散得到的 $p_t(\boldsymbol{x})$ 相等．

为了保持 SDE 的一般形式，这里的 $\overline{\boldsymbol{W}}$ 是反向的 Wiener 过程．不过我们可以无视掉这个区别，因为 $\mathrm{d}\overline{\boldsymbol{W}}$ 仍然是一样的无穷小高斯噪声 $\mathcal{N}(\boldsymbol{0},\boldsymbol{I}\mathrm{d}t)$．

有了保持分布的反向 SDE，我们就可以从 $p_T(\boldsymbol{x})$ 的样本开始进行反向扩散，从而对 $p_0(\boldsymbol{x})$ 进行采样了，如下图所示．观察反向扩散 SDE，可以发现它的漂移项多了一项 $g(t)^2\nabla_{\boldsymbol{x}}p_t(\boldsymbol{x})$，这项修正保证分布始终保持一致．可以发现，这里面的 $\nabla_{\boldsymbol{x}}p_t(\boldsymbol{x})$ 就是我们前面介绍的分数．也就是说，SDE 也是从分数的角度去看扩散模型的．SDE 只是提供了一个描述扩散过程的数学框架，但是模型中真正的学习数据的部分仍然是我们在 3.2.2 中介绍的分数匹配的神经网络．



本图来自 SDE 论文的原图．

4.1.3 从 SDE 的视角看 DDPM 和 NCSN

有了 SDE 的理论框架，我们再来回头重新看一下前面介绍的 DDPM 和 NCSN 模型.

NCSN

由于 NCSN 在 SDE 的框架下稍更简单一些，所以我们先来看 NCSN. 在 NCSN 中，我们以 $\tilde{\boldsymbol{x}} = \boldsymbol{x} + \sigma\boldsymbol{\epsilon}$ 的方式向数据加入不同强度的噪声. 我们把它换一种记号，记作 $\boldsymbol{x}_i = \boldsymbol{x}_0 + \sigma_i\boldsymbol{\epsilon}$. 如果我们把噪声看作一点一点注入的，则可以化为递推公式 $\boldsymbol{x}_i = \boldsymbol{x}_{i-1} + \sqrt{\sigma_i^2 - \sigma_{i-1}^2}\boldsymbol{\epsilon}_{i-1}$. 将它连续化可以得到

观察 NCSN 注入的噪声可以发现，当 $t \to \infty$ 时，$\boldsymbol{x}$ 的方差是发散的，因此 NCSN 的 SDE 也被称作 Variance Exploding (VE) SDE. 而下面的 DDPM 则保持方差始终为 1，因此称为 Variance Preserving (VP) SDE.

$$\mathrm{d}\boldsymbol{x} = \sqrt{\frac{\mathrm{d}(\sigma^2(t))}{\mathrm{d}t}}\mathrm{d}\boldsymbol{W} \tag{4.6}$$

这就是 NCSN 的正向 SDE. 对比式 4.4，可以发现 NCSN 的漂移系数 $\boldsymbol{f}(\boldsymbol{x}, t) = 0$，扩散系数 $g(t) = \sqrt{\mathrm{d}(\sigma^2(t))/\mathrm{d}t}$. 知道了这两个系数，我们就可以根据式 4.5 直接写出反向扩散的 SDE

$$\mathrm{d}\boldsymbol{x} = -\frac{\mathrm{d}(\sigma^2(t))}{\mathrm{d}t}\nabla_{\boldsymbol{x}}\log p_t(\boldsymbol{x})\mathrm{d}t + \sqrt{\frac{\mathrm{d}(\sigma^2(t))}{\mathrm{d}t}}\mathrm{d}\overline{\boldsymbol{W}} \tag{4.7}$$

可以看到，反向扩散 SDE 的漂移系数是沿着分数 $\nabla_{\boldsymbol{x}}\log p_t(\boldsymbol{x})$ 的反方向的. 那么当我们将时间 t 从 T 到 0 反向移动时，$\boldsymbol{x}$ 便会沿着分数的方向漂移，并加上一些随机噪声，这就是 Langevin 采样. 同时，这里的 $\nabla_{\boldsymbol{x}}\log p_t(\boldsymbol{x})$ 实际上是带有一定的噪声的分布 $p_t(\boldsymbol{x})$ 的分数. 在 t 变小的过程中，噪声也逐渐变小，而这实际上就是我们学过的退火 Langevin 采样! 因此，我们从 SDE 的角度，给定了正向扩散方程 4.6，就直接得到了整个 NCSN 的架构.

DDPM

对于 DDPM，我们也可以以同样的方式将它化为 SDE 形式. DDPM 的正向扩散方程为 $\boldsymbol{x}_i = \sqrt{1-\beta_i}\boldsymbol{x}_{i-1} + \sqrt{\beta_i}\boldsymbol{\epsilon}_i$. 将它连续化时，每一小步的 $\beta_t \ll 1$，从而可以得到

这里的 $-\frac{1}{2}\beta$ 来自 $\sqrt{1-\beta} - 1$ 的一阶近似.

$$\mathrm{d}\boldsymbol{x} = -\frac{1}{2}\beta(t)\boldsymbol{x}\mathrm{d}t + \sqrt{\beta(t)}\mathrm{d}\boldsymbol{W} \tag{4.8}$$

可以观察到它的漂移系数系数为 $\boldsymbol{f}(\boldsymbol{x}, t) = -\frac{1}{2}\beta(t)\boldsymbol{x}$，扩散系数为 $g(t) = \sqrt{\beta(t)}$. 这样，它的反向扩散 SDE 为

$$\mathrm{d}\boldsymbol{x} = \left(-\frac{1}{2}\beta(t)\boldsymbol{x} - \beta(t)\nabla_{\boldsymbol{x}}\log q_t(\boldsymbol{x})\right)\mathrm{d}t + \sqrt{\beta(t)}\mathrm{d}\overline{\boldsymbol{W}} \tag{4.9}$$

很容易证明，这个反向 SDE 的漂移部分就是式 3.6 的连续化. 因此，我们也就由 SDE 框架得到了 DDPM. 由这两个例子可以看出，NCSN 和 DDPM 只是两个对漂移和扩散项定义略有不同的 SDE.

由 SDE 的角度也更容易解释学习 DDPM 的一些常见问题——例如，用神经网络估计出了 $\bar{\boldsymbol{\epsilon}}_{t-1}$ 后，为什么不可以用式 3.2 直接采样 $\boldsymbol{x}_0$? 在 DDPM 的反向扩散的视角下，我们只能说这样估计的误差可能会比较大，但难以给出直观的解释. 但从 SDE 的角度这个问题就非常简单了——我们估计的 $\bar{\boldsymbol{\epsilon}}_{t-1}$ 实际上是反向 SDE 的漂移项，它只是对噪声的一种“局部估计”. 我们向着我们估计的噪声的反方向走一小步，相当于做了一小步离散化的 SDE 演化. 但我们不能期望沿着这方向直着走一大步就能直接到达 $\boldsymbol{x}_0$. 再例如，为什么 DDPM 的反向扩散一定要加上合适大小的噪声，而不能直接用估计的均值? 从 SDE 的角度我们可以说，如果把噪声从反向 SDE 中去掉，那么反向扩散所决定的分布 $p_t(\boldsymbol{x})$ 就

于正向过程的分布不再相等，我们最终得到的 $\boldsymbol{x}_0$ 就不再是对 $p_0(\boldsymbol{x})$ 的一个采样．这一点我们在下一节会有更直接的认识．

4.2 概率流常微分方程

4.2.1 从 SDE 到 ODE

我们前面使用 SDE 的框架重新解释了 DDPM 和 NCSN．不过，我们在 3.1.4 中还讨论了一种和 DDPM 高度相关的模型，称为 DDIM．那么 DDPM 和 DDIM 的关系又该如何纳入 SDE 的框架中呢？在 SDE 的基础上，人们进一步发展了一个解释确定性扩散模型的框架．下面我们就来看一下这个框架．

回顾一下我们在 3.1.4 中讨论的，DDIM 找到了一族新的正向扩散的分布，使得边际分布 $q(\boldsymbol{x}_t|\boldsymbol{x}_0)$ 保持不变．我们可以把这个思路用到 SDE 中，也就是找到一族 SDE，使得它所决定的边际分布 $p(\boldsymbol{x}_t|\boldsymbol{x}_0)$ 与式 4.4 相同．可以证明，这一族 SDE 可以表示为

$$\mathrm{d}\boldsymbol{x} = \left[\boldsymbol{f}(\boldsymbol{x},t) - \frac{1}{2}\left(g(t)^2 - \sigma(t)^2\right)\nabla_{\boldsymbol{x}}\log p_t(\boldsymbol{x})\right]\mathrm{d}t + \sigma(t)\mathrm{d}\boldsymbol{W}$$

其中 $\sigma(t)$ 是任意一个函数．当 $\sigma(t) = g(t)$ 时，我们就回到了式 4.4．而如果我们在这里令 $\sigma(t) = 0$，那么扩散项完全消失，SDE 退化为一个常微分方程 (ordinary differential equation, ODE)

$$\mathrm{d}\boldsymbol{x} = \left[\boldsymbol{f}(\boldsymbol{x},t) - \frac{1}{2}g(t)^2\nabla_{\boldsymbol{x}}\log p_t(\boldsymbol{x})\right]\mathrm{d}t \tag{4.10}$$

这个方程称为式 4.4 的 SDE 所对应的 **概率流 ODE** (Probability Flow ODE)．可以看到，直接将 SDE 的扩散项去掉是不能得到一个保持分布的方程的．要保持分布相同，我们需要对漂移项做一定的修改，而修正项的方向仍然是我们熟悉的分数 $\nabla_{\boldsymbol{x}}\log p_t(\boldsymbol{x})$．

以特例理解扩散项对于分布的影响，我们可以考虑一个 SDE $\mathrm{d}x = \mathrm{d}W$，这是一个纯扩散方程，分布最终会变为高斯噪声．而如果我们直接去掉扩散项，变为 $\mathrm{d}x = 0$，那么 x 的分布就不会随着时间变化．

ODE 决定了一个确定的演化路径，因此它的逆向方程不需要任何修正，就是它自己．因此，我们使用同一个 ODE 从 $p_T(\boldsymbol{x})$ 反向演化，就可以得到反向扩散过程．

将带有随机扩散的 SDE 变为确定性的 ODE 由许多好处．我们前面在 DDIM 的小节已经强调了，确定性的模型可以跳步以加速生成过程．但概率流 ODE 的优势还不仅如此．确定性的模型给每个样本赋予了一个确定的、独一无二的隐变量表示．由于隐变量常常代表图像的一些高层特征，这使得我们可以在隐变量空间里对图像进行处理、修改．最后，概率流 ODE 可以直接用于计算数据的似然，其计算方法如下．

$$\log p_0(\boldsymbol{x_0}) = \log p_T(\boldsymbol{x}_T) - \int_0^T \nabla_{\boldsymbol{x}_t}\cdot\boldsymbol{f}(\boldsymbol{x}_t,t)\mathrm{d}t$$

其中 $\nabla_{\boldsymbol{x}_t}\cdot\boldsymbol{f}(\boldsymbol{x}_t,t)$ 是时间为 t 时向量场 $\boldsymbol{f}(\boldsymbol{x})$ 的散度．$\boldsymbol{f}(\boldsymbol{x})$ 越是向四周散开的地方，它的散度越大．而逆向 ODE 是沿着 $-\boldsymbol{f}(\boldsymbol{x})$ 的方向走的，所以直观地看，$-\boldsymbol{f}(\boldsymbol{x})$ 越“汇聚”，此处的似然就应该越大．因此，$-\boldsymbol{f}(\boldsymbol{x})$ 描述了演化过程中似然的增加量．这个式子说明，我们只需从某个起点逆着时间进行概率流 ODE 决定的演化，并加上演化过程中似然的增加量，就得到了最终的数据似然．

4.2.2 从 ODE 的视角看 DDIM

DDIM 作为一个确定性的扩散模型，应该可以从概率流 ODE 的视角来解释．如果我们把 DDIM 的递推公式 3.10 变一下形，可以得到

$$\frac{\boldsymbol{x}_t}{\sqrt{\bar{\alpha}_t}} - \frac{\boldsymbol{x}_{t-1}}{\sqrt{\bar{\alpha}_{t-1}}} = \left(\sqrt{\frac{1-\bar{\alpha}_t}{\bar{\alpha}_t}} - \sqrt{\frac{1-\bar{\alpha}_{t-1}}{\bar{\alpha}_{t-1}}}\right)\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t,t)$$

那么它的连续化版本就应该是

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\boldsymbol{x}_t}{\sqrt{\bar{\alpha}_t}}\right)=\frac{\mathrm{d}}{\mathrm{d}t}\left(\sqrt{\frac{1-\bar{\alpha}_t}{\bar{\alpha}_t}}\right)\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t,t) \tag{4.11}$$

这就是 DDIM 的概率流 ODE. 注意这里的 $\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t,t)$ 是一个确定的函数而不是噪声，因此这是一个 ODE 而非 SDE.

另一方面，由于 DDIM 是与 DDPM 的边际分布一样的确定性模型，那么 DDIM 的 ODE 就应该是 DDPM 的 SDE 所对应的概率流 ODE. 我们将式 4.9 改成如 4.10 的概率流 ODE，可以得到

$$\mathrm{d}\boldsymbol{x}=-\frac{1}{2}\beta(t)(x+\nabla_{\boldsymbol{x}}\log p_t(\boldsymbol{x}))\mathrm{d}t \tag{4.12}$$

可以证明，4.12 与 4.11 是等价的. 这样，我们就将 DDIM 也纳入了微分方程的框架中，并可以看到相同边际分布的随机模型和确定性模型的确就是 SDE 与概率流 ODE 的关系.

证明这个关系需要注意到 $\mathrm{d}\bar{\alpha}_t/\mathrm{d}t=-\bar{\alpha}_t(1-\alpha_t)$ 和式 3.15.

4.3 微分方程求解器

SDE 和概率流 ODE 为扩散模型提供了统一的理论框架，让我们可以更好地理解、设计扩散模型. 不过，SDE 和 ODE 都是连续的微分方程，而计算机能直接处理的必须是离散的差分方程. 例如，我们在 4.2.1 看到了 NCSN 和 DDPM 就是 SDE 的两种差分形式. 将微分方程化为差分方程从而让计算机可以求解的方式称为微分方程的 **求解器** (solver).

4.3.1 ODE 求解器

在讨论 SDE 之前，我们首先来看一下如何将更为简单的 ODE 化为差分方程. 对于一个一般的一阶 ODE，例如 $\mathrm{d}\boldsymbol{x}=\boldsymbol{f}(\boldsymbol{x},t)\mathrm{d}t$，最直接的方法显然是将它化为

$$\boldsymbol{x}_{t+\Delta t}=\boldsymbol{x}_t+\boldsymbol{f}(\boldsymbol{x}_t,t)\Delta t \tag{4.13}$$

这种求解器称为 **Euler 法**. Euler 法容易理解、便于实现，但精确度仍有提升的空间. 这是因为，Euler 法实际上是使用 $\boldsymbol{x}_t$ 处的切线近似 $\boldsymbol{x}_t$ 的变化，但在 Δt 较大时切线对于曲线的偏离会比较大. 因此，我们可以对这个切线的斜率进行一定的修正，例如取起始点的切线斜率的平均值作为最终斜率的近似. 也就是

$$\boldsymbol{x}_{t+\Delta t}=\boldsymbol{x}_t+\frac{1}{2}(\boldsymbol{f}(\boldsymbol{x}_t,t)+\boldsymbol{f}(\boldsymbol{x}_{t+\Delta t},t+\Delta t))\Delta t \tag{4.14}$$

这种求解器称为 **Heun 法**. 这种对斜率的修正方式还可以继续被推广. 例如，一种很常见的求解器称为四阶 **Runge-Kutta 法** (RK4)，它就是在 $[t,t+\Delta t]$ 区间内以特定方式取了四个点，并取它们的斜率的加权平均作为最终斜率. 这些更高级的求解器在 Δt 较大时比 Euler 法更加精确.

4.3.2 SDE 求解器

SDE 求解器与 ODE 求解器非常类似，只是需要加上噪声作为扩散项. 对于扩散模型中的一阶 SDE，例如 $\mathrm{d}\boldsymbol{x}=\boldsymbol{f}(\boldsymbol{x},t)\mathrm{d}t+g(t)\mathrm{d}\boldsymbol{W}$，最直接的离散化方式显然是

$$\boldsymbol{x}_{t+\Delta t}=\boldsymbol{x}_t+\boldsymbol{f}(\boldsymbol{x}_t,t)\Delta t+g(t)\sqrt{\Delta t}\boldsymbol{\epsilon} \tag{4.15}$$

其中 $\boldsymbol{\epsilon}$ 是一个标准高斯噪声. 这其实就是 ODE 的 Euler 法加上了噪声项，称为 **Euler-Maruyama 法**，这也是 SDE 最为基础的求解器.

与 ODE 中的 Euler 法类似，Euler-Maruyama 法也可以通过修正来提升精确度．例如一种很重要的 SDE 求解器称为 **Milstein** 法，它的更新公式为

$$\boldsymbol{x}_{t+\Delta t} = \boldsymbol{x}_t + \boldsymbol{f}(\boldsymbol{x}_t, t)\Delta t + g(t)\sqrt{\Delta t}\boldsymbol{\epsilon} + \frac{1}{2}g(t)g'(t)(\boldsymbol{\epsilon}^2 - \Delta t) \tag{4.16}$$

可以看到，Milstein 法增加了对于扩散项的一项修正．类似地，SDE 中也有随机 RK4 算法，但其公式较为复杂，我们在此就不展开了．

与 ODE 求解器类似，高级的 SDE 求解器也可以增加步长、提高精确度．同时，这一小节的高级 SDE 求解器都只是对扩散项进行修正．我们还可以同时用 ODE 求解器对于漂移项进行修正，以使两项都达到更高的精度．

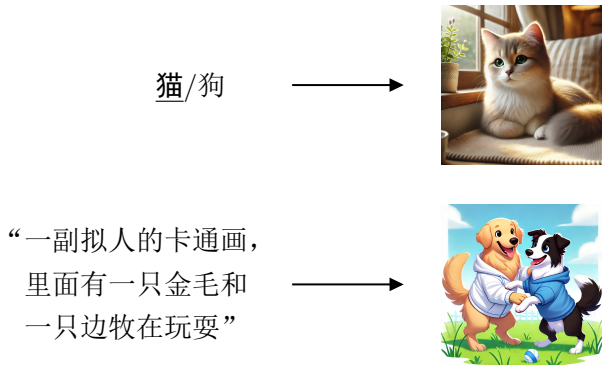
5　进阶技术

我们前面几章已经讲完了扩散模型的所有基本原理．这一章，我们来讨论实现扩散模型时常常会使用到的一些稍微进阶一些的技术．当然，由于扩散模型是一个较为年轻但已经被广泛使用的模型，因此本章介绍的内容虽然都来自几年前，但已经成为了被广泛使用的基础技术．若想了解真正最前沿的进阶技术，请阅读最新的文献．

5.1　条件扩散模型

5.1.1　条件生成模型的基本概念

在前面的章节中，我们讨论的都是用一个模型去生成一类图像——例如猫猫图片．但是我们使用到的生成式模型大多都可以根据指令生成不同的东西．例如，一个简单的模型可能会让我们输入“猫” 或者“狗”，从而为我们生成一张猫或者狗的图像．更为复杂的模型可以接受一段描述性的文字作为输入，据此生成一张符合描述的图像．这类生成式模型称为 **条件生成模型** (conditional generative model)．



从统计的角度看，条件生成模型仍然是要从一个分布里采样，只不过这个分布是以我们的输入为条件的．按照机器学习的惯例，我们将这个条件记作 y，那么我们要采样的就是条件分布 $p(\boldsymbol{x}|y)$．那么同样，我们也可以使用扩散模型来实现这一采样．这一节，我们就来看一下如何将条件加入已有的扩散模型算法中．我们在这里将不限于某一种具体的算法如 DDPM 或 NCSN，而是从基于分数的角度，来将条件加入整个扩散模型的共有框架中．

5.1.2　分类器引导与无分类器生成

在条件生成模型中，一切都是基于条件 y 的，我们的目标是从条件分布 $p(\boldsymbol{x}|y)$ 中进行采样．所以从基于分数的角度来看，我们需要一个条件分数 $\nabla_{\boldsymbol{x}_t}\log p(\boldsymbol{x}_t|y)$ 用于条件采样．为了和前面的无条件扩散模型联系起来，我们可以将条件分数做一个简单的变形，得到

$$\begin{aligned}\nabla_{\boldsymbol{x}_t}\log p(\boldsymbol{x}_t|y) &= \nabla_{\boldsymbol{x}_t}\log\left(\frac{p(\boldsymbol{x}_t)p(y|\boldsymbol{x}_t)}{p(y)}\right)\\ &= \underbrace{\nabla_{\boldsymbol{x}_t}\log p(\boldsymbol{x}_t)}_{\text{非条件分数}} + \underbrace{\nabla_{\boldsymbol{x}_t}\log p(y|\boldsymbol{x}_t)}_{\text{分类器梯度}} - \underbrace{\nabla_{\boldsymbol{x}_t}\log p(y)}_{=0}\end{aligned} \tag{5.1}$$

可以看到，这里我们将条件分数化为了两项，其中第一项就是我们熟悉的非条件分数，它由所有的数据训练而来．而在第二项中，$p(y|\boldsymbol{x}_t)$ 就是一个传统的概率性分类器．而

$\nabla_{\boldsymbol{x}_t} \log p(y|\boldsymbol{x}_t)$ 就是这个分类器输出的对数概率对输入 $\boldsymbol{x}_t$ 的梯度．也就是说，我们在这里需要实现一个分类器，这个分类器可以将带有噪声的图像进行分类．因此，这种条件生成模式称为 **分类器引导的** (classifier-guided) 扩散模型． 通常，我们也使用一个神经网络实现这个分类器，而梯度可以通过反向传播进行计算．

反向传播通常被用于计算损失函数对于模型参数的梯度，但这里是计算输出的对数概率对于输入的梯度．

实际使用时，人们发现分类器引导的强度会影响输出的多样性——引导强度越高，不同类别的输出差异越大，但类别内的多样性会越小．因此，为了调整分类器引导的强度，人们常常在这里添加一个超参数 w，从而使用的分数变为

$$\nabla_{\boldsymbol{x}_t} \log p(\boldsymbol{x}_t|y) = \nabla_{\boldsymbol{x}_t} \log p(\boldsymbol{x}_t) + w\nabla_{\boldsymbol{x}_t} \log p(y|\boldsymbol{x}_t) \tag{5.2}$$

而另一类重要的加入条件的方法称为 **无分类器** (classifier-free) 的扩散模型．无分类器模型求解条件分数的方法更加直接，不需要训练单独的一个分类器．它直接使用一个神经网络 $\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t, y)$ 拟合条件分数．很容易证明，上面式 5.2 的加权条件分数可以在无分类器方法中被化为

$$\tilde{\boldsymbol{s}}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t, y) = (w+1)\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t, y) - w\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t) \tag{5.3}$$

其中 $\boldsymbol{s}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)$ 是非条件分数．为了降低训练开销并增加泛化性能，这里的条件分数和非条件分数使用同一个网络进行训练．在训练时，我们可以随机地将条件 y 置空，用于训练非条件分数．这样，在使用网络时，我们也只需将 y 置空便可以得到非条件分数．

分类器引导的算法和无分类器算法各有优劣——分类器引导的算法训练较为容易，因为和非条件情况相比，我们只需要多训练一个分类器，而分类器显然更容易实现．不过，在采样时，分类器引导的算法需要对分类器进行反向传播以计算梯度，因此采样速度更慢．而无分类器的算法需要训练一个更复杂的网络以估计条件分数，因此训练更慢，但采样更快．总体来说，无分类器引导的算法性能更好． 同时，无分类器算法还有另一个优势，那就是条件 y 不需要是分类变量，它完全可以是连续变量，如文字编码等，因此可以用于更复杂的文字转图像任务．因此，现在主流算法都使用无分类器算法．

所谓 CLIP 引导就是使用 CLIP 文字编码的无分类器引导，它被用于 DALL·E 2 和 Stable Diffusion．但也有使用其他文字编码引导的算法．

5.2 加速生成

模型加速是深度学习里的一个重要话题，而在扩散模型中尤为如此．这是因为扩散模型往往需要许多次递归（即反向扩散）才能完成采样，速度往往较慢．因此，模型加速是扩散模型的一个核心需求．我们在前面的章节中已经零散地讨论了一些模型加速的方式，例如降低方差以跳步采样（如 DDIM）、选择高阶求解器等．本节我们来讨论几种更进阶的加速方法．

5.2.1 知识蒸馏与一致性模型

我们要介绍的第一类加速方法与 DDIM 的思路类似，都是减少反向扩散的步数．这类方法使用了深度学习里的一种重要技术，称为 **知识蒸馏** (knowledge distillation)．知识蒸馏是指将一个已经训练好的模型，即“教师模型” (teacher model) 的知识迁移到一个新的、一般较小的网络，即“学生模型” (student model) 中，从而降低计算量．将知识蒸馏运用在扩散模型中，目前仍然是一个活跃的科研领域．我们在这里只介绍较为有名的两种方法——渐进蒸馏和一致性模型．

渐进蒸馏 (progressive distillation) 是指通过知识蒸馏的方式将多步反向扩散合并为一步的过程．考虑一个确定性的反向扩散算法，例如我们熟悉的 DDIM．为了简化模型，我

为了防止出现的字母 $\boldsymbol{x}$ 过多，我们在渐进蒸馏这部分将扩散的所有中间步骤都改为字母 $\boldsymbol{z}$，只有原始图像记作 $\boldsymbol{x}$．除此之外，这里的记号保持和前面 DDIM 的相同，和渐进蒸馏的论文并不一致．

们将 DDIM 做一些改动， 使用一个神经网络 $\hat{\boldsymbol{x}}_{\boldsymbol{\eta}}(\cdot)$ 从带噪声的图像 $\boldsymbol{z}_t$ 预测去噪后的 $\boldsymbol{x}$．那么反向扩散式 3.10 可以化为

$$\boldsymbol{z}_{t-1} = \sqrt{\bar{\alpha}_{t-1}}\hat{\boldsymbol{x}}_{\boldsymbol{\eta}}(\boldsymbol{z}_t) + \sqrt{\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}}\left(\boldsymbol{z}_t - \sqrt{\bar{\alpha}_t}\hat{\boldsymbol{x}}_{\boldsymbol{\eta}}(\boldsymbol{z}_t)\right) \tag{5.4}$$

由 $\boldsymbol{z}_{t-1}$ 我们可以再进行一步相同的反向扩散，得到 $\boldsymbol{z}_{t-2}$．之后，我们希望训练一个新的神经网络 $\tilde{\boldsymbol{x}}_{\boldsymbol{\theta}}(\cdot)$，使得我们使用 $\tilde{\boldsymbol{x}}_{\boldsymbol{\theta}}(\boldsymbol{z}_t)$ 代替 $\hat{\boldsymbol{x}}_{\boldsymbol{\eta}}(\boldsymbol{z}_t)$ 用上式进行反向扩散的时候，会直接得到 $\boldsymbol{z}_{t-2}$．这样，用一步 $\tilde{\boldsymbol{x}}_{\boldsymbol{\theta}}(\cdot)$ 就可以做到用原来的 $\hat{\boldsymbol{x}}_{\boldsymbol{\eta}}(\boldsymbol{z}_t)$ 两步的效果．根据这个关系，很容易求出 $\tilde{\boldsymbol{x}}_{\boldsymbol{\theta}}(\boldsymbol{z}_t)$ 的目标值应为

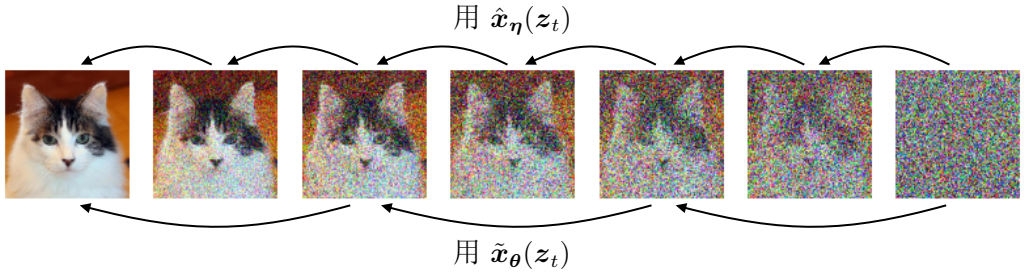
$$\boldsymbol{x}_t^{\text{target}} = \frac{\boldsymbol{z}_{t-2} - \sqrt{\frac{1-\bar{\alpha}_{t-2}}{1-\bar{\alpha}_t}}\boldsymbol{z}_t}{\sqrt{\bar{\alpha}_{t-2}} - \sqrt{\frac{1-\bar{\alpha}_{t-2}}{1-\bar{\alpha}_t}}\sqrt{\bar{\alpha}_t}} \tag{5.5}$$

这样，我们训练新网络 $\tilde{\boldsymbol{x}}_{\boldsymbol{\theta}}(\cdot)$ 的损失函数应为

$$L = \mathbb{E}_t\left[w(\bar{\alpha}_t)||\tilde{\boldsymbol{x}}_{\boldsymbol{\theta}}(\boldsymbol{z}_t) - \boldsymbol{x}_t^{\text{target}}||^2\right] \tag{5.6}$$

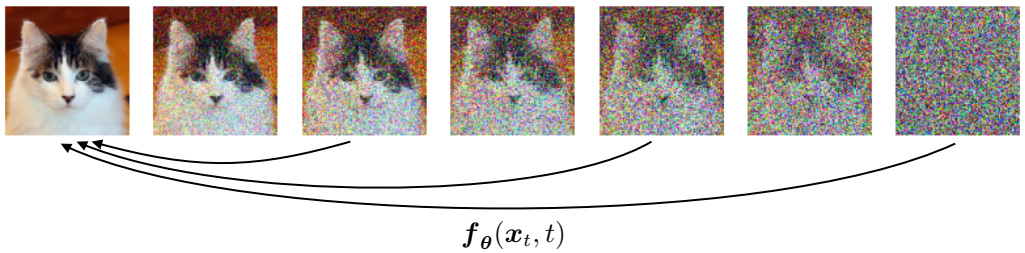
权重常选择为 $w(\bar{\alpha}_t) = (1-\bar{\alpha}_t)^{-1}$.

其中 $w(\bar{\alpha}_t)$ 是权重．这样，我们使用新的网络 $\tilde{\boldsymbol{x}}_{\boldsymbol{\theta}}(\cdot)$ 进行反向扩散时，一步相当于原来的两步，那么总步数减半即可生成图像．如果我们再做一次类似的蒸馏，那么总步数可以再次减半．这样多次蒸馏以逐渐降低步数，即为渐进蒸馏．当然，过度的蒸馏会伴随着图像生成质量的损失．通常来说，渐进蒸馏可以将反向扩散步骤降低至 10 步左右．



既然我们可以通过知识蒸馏将反向扩散降低至很少的步数，那么能不能直接把它降低到一步，直接求得 $\boldsymbol{x}_0$ 呢？OpenAI 开发了一种新的算法，称为 **一致性模型** (consistency model)，其目标就是得到一个这样的一步模型．具体来说，一致性模型使用连续的、确定性的扩散过程，也就是概率流 ODE． 它希望找到一个满足“一致性” 的函数 $\boldsymbol{f}(\boldsymbol{x}, t)$，使得扩散路径上的每一个点 $\boldsymbol{x}_t$ 都直接被映射回该路径上接近起点处的 $\boldsymbol{x}_\epsilon$，其中 ϵ 是一个很小的数．

这里希望映射回 $\boldsymbol{x}_\epsilon$ 而非 $\boldsymbol{x}_0$ 是为了数值稳定性.



这个一致性函数需要满足边界条件 $\boldsymbol{f}(\boldsymbol{x}_\epsilon, t) = \boldsymbol{x}_\epsilon$ 对任意路径上的 $\boldsymbol{x}_\epsilon$ 都成立．那么我们可以选择如下形式的 $\boldsymbol{f}(\boldsymbol{x}_\epsilon, t)$

$$\boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t) = c_{\text{skip}}(t)\boldsymbol{x}_t + c_{\text{out}}(t)\boldsymbol{F}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t) \tag{5.7}$$

其中 $c_{\text{skip}}(t)$ 和 $c_{\text{out}}(t)$ 是指定的两个可微函数，满足 $c_{\text{skip}}(\epsilon)=1$，$c_{\text{out}}(\epsilon)=0$，而 $\boldsymbol{F}_{\boldsymbol{\theta}}(\boldsymbol{x}_t,t)$ 是一个神经网络．我们只需要训练这个神经网络，就可以得到一致性模型了．

一致性模型论文中选择的两个函数是

$$c_{\text{skip}}(t)=\frac{\sigma_0^2}{(t-\epsilon)^2+\sigma_0^2}$$

$$c_{\text{out}}(t)=\frac{\sigma_0(t-\epsilon)}{\sqrt{t^2+\sigma_0^2}}$$

其中 $\sigma_0=0.5$ 是数据的标准差．

训练一致性模型有两种方法，第一种是从已经训练好的扩散模型蒸馏而来，称为一致性蒸馏 (consistency distillation)．在一致性蒸馏中，我们希望最小化 $\boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{x}_t,t)$ 与 $\boldsymbol{x}_\epsilon$ 的差异，但计算 $\boldsymbol{x}_\epsilon$ 需要完成整个反向扩散过程．若要对每个可能的路径都进行完整的反向扩散，这显然是不可能的．因此，一致性蒸馏选择了一种间接的训练目标，即让路径上相邻两点的输出尽量相似．这样，一致性蒸馏的损失函数为

$$L=\mathbb{E}_{\boldsymbol{x}_0,i,\boldsymbol{x}_{i+1}}\left[||\boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{x}_{t_{i+1}},t_{i+1})-\boldsymbol{f}_{\boldsymbol{\theta}}(\hat{\boldsymbol{x}}_{t_i},t_i)||^2\right] \tag{5.8}$$

其中 $\hat{\boldsymbol{x}}_{t_i}$ 是用我们已有的扩散模型由 $\boldsymbol{x}_{t_{i+1}}$ 求出的．注意到以式 5.7 定义的 $\boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{x}_t,t)$ 的形式保证了模型会映射到各自路径的起始点上，而不会将所有的输入映射为同一点．

真正的两种一致性模型的损失函数比这里介绍的要复杂一些，但主要复杂在一些技术细节上，这里保留了主要思想．

而第二种方式则是直接从零训练出一个一致性模型，称为一致性训练 (consistency training)．一致性训练中没有训练好的神经网络让我们做反向扩散，不过其作者发现其实有一种很简单的方式来估计分数以进行反向扩散．如果我们使用一种比较简单的正向扩散 SDE，即 $\mathrm{d}\boldsymbol{x}=\sqrt{2t}\mathrm{d}\boldsymbol{W}$．那么很容易证明，此时分数可以表示为

$$\nabla_{\boldsymbol{x}}\log p_t(\boldsymbol{x})=\mathbb{E}\left[-\frac{\boldsymbol{x}_0-\boldsymbol{x}_t}{t^2}\middle|\boldsymbol{x}_t\right]$$

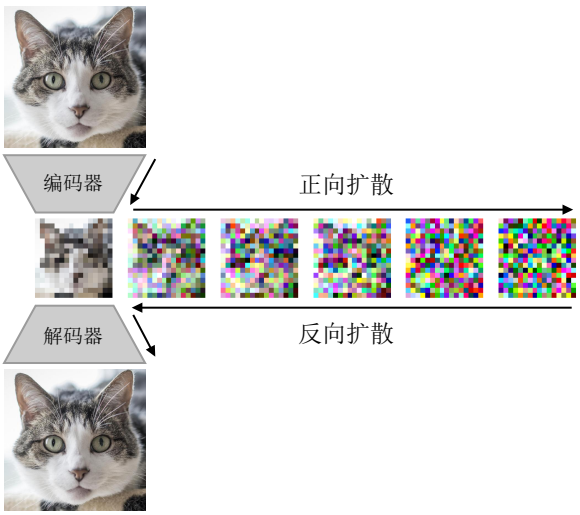
由于这种情况下 $\boldsymbol{x}_t\sim\mathcal{N}(\boldsymbol{x}_0,t^2\boldsymbol{I})$，那么如果我们把 $\boldsymbol{x}_{t_{i+1}}$ 写成 $\boldsymbol{x}_0+t_{i+1}\boldsymbol{z}$，进而就很容易推出反向扩散一步的 $\hat{\boldsymbol{x}}_{t_i}$ 为 $\boldsymbol{x}_0+t_i\boldsymbol{z}$．这样，一致性蒸馏的损失函数可以被化为

$$L=\mathbb{E}_{\boldsymbol{x}_0,i,\boldsymbol{z}}\left[||\boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{x}_0+t_{i+1}\boldsymbol{z},t_{i+1})-\boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{x}_0+t_i\boldsymbol{z},t_i)||^2\right] \tag{5.9}$$

这就是一致性训练的损失函数．目前看，几种蒸馏算法的生成图像的质量比较为：原始的扩散模型 > 一致性蒸馏 > 渐进蒸馏 > 一致性训练，所以可以看到这几种蒸馏算法都是用速度换取了质量．不过，蒸馏后的模型还是可以用来做多步的反向扩散，从而重新用速度换取质量的．

5.2.2 潜在扩散模型

另一种重要的加速方法称为 **潜在扩散模型** (Latent Diffusion Model, LDM)，它是 Stable Diffusion 的核心算法．



LDM 加速的思想是将图像压缩到比较低维之后再训练扩散模型．具体来说，LDM 使用自编码器将图像降低到低维的隐变量空间中． 这个隐变量空间的维数通常不会太低，例如 Stable Diffusion 的隐变量维数是 $4 \times 64 \times 64$，因此隐变量通常会看起来类似于压缩的、低像素的图像． LDM 使用这些隐变量训练一个扩散模型，那么这个扩散模型自然也就可以用于生成一个合理的隐变量．这个生成的隐变量再通过自编码器的解码器部分，就会生成一张高分辨率的图像．

许多地方会写 LDM 使用 VAE 进行降维，这是不准确的，它只使用了一个传统的自编码器结构．

5.3　图像处理的其他应用

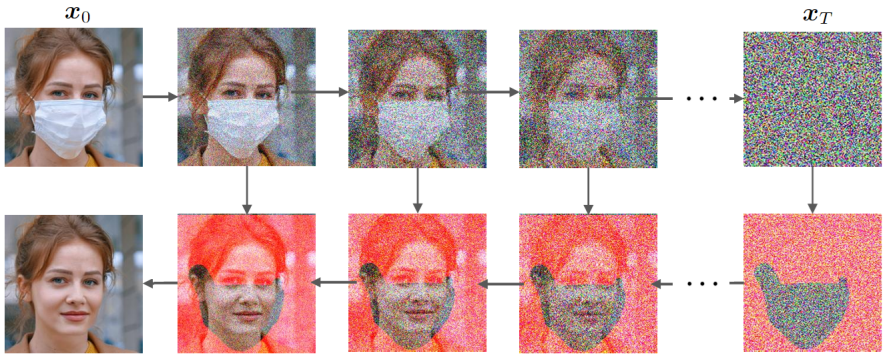
我们前面都在讨论扩散模型的最原始任务——从零生成一个图像．这一小节，我们来讨论一下扩散模型如何用于图像生成相关的其他任务．由于扩散模型还是个比较新的领域，我们在这里只介绍几个比较基础的任务中比较著名的算法．虽然也有算法将扩散模型用于更为复杂的任务中，但目前还没有表现得特别突出，因此在这就先不介绍了．

5.3.1　图像修复

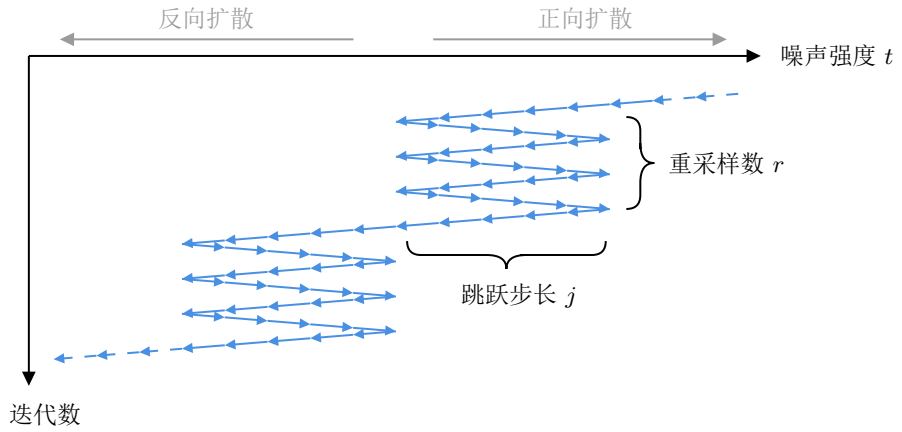
和图像生成相关的一个非常经典的任务是图像修复，也就是给定一个部分被遮挡的图像，让算法去填充被遮挡的部分．当然，图像修复算法不止可以用于修复图像，它还可用于去除图像中的一些部分，例如从照片中去除路人等．

我们在这里要介绍的是一个基于 DDPM 的非常著名的图像修复算法——RePaint．RePaint 的输入是一张带有遮挡的图像 $\boldsymbol{x}_0$ 和遮挡的 **掩膜** (mask) $\boldsymbol{m}$．前者在被遮挡的区域的像素值可以是 0 或任意值，因为我们不会用到它．而后者是一个和图像一样大小的二维数组，在被遮挡处为 0，未被遮挡处为 1，它用来指示未被遮挡的部分在哪．

RePaint 的最基本的思想如下图所示．它使用一个训练好的普通 DDPM 模型，从高斯噪声反向扩散生成图像．为了使生成的图像在非遮挡区域与提供的图像一致，我们在每一步都将非遮挡部分用输入的已知图像替换．当然，为了保证噪声强度一致，我们替换的也需要是加入了等量噪声的已知图像．这样，在反向扩散的过程中，已知的部分可以引导反向扩散的方向，从而得到合理的生成图像．



不过，这样做生成的图像效果还是不太好．被遮挡的部分往往低级特征与周围相符，但高级特征却不合理．也就是说，在反向扩散的过程中，未知部分没有吸收到足够多的来自已知部分的信息．为了解决这一问题，RePaint 引入了重采样 (resampling) 的方法——对于每个已经替换了已知部分的 $\boldsymbol{x}_t$，我们可以把它再次正向扩散几步到 $\boldsymbol{x}_{t+j}$，再反向扩散回 $\boldsymbol{x}_t$，如此往复几次，每次反向扩散时未知部分就会再吸收一些来自已知部分的信息．这样，最终未知部分就会吸收到足够多的信息，生成更有意义的图像．整个过程如下图所示．注意每次反向扩散后，图像的已知部分都会被替换．



实验发现，重采样数 r 可以显著增加生成部分与已知部分的一致性，而仅仅增加扩散步数 T 则不能．跳跃步长 j 似乎也会提高生成的质量．不过这种重采样方式极大的增加了算法所需要的扩散步数，因此 RePaint 生成图像的速度是非常慢的．

5.3.2　Image-to-Image 转换

RePaint 通过精心的算法设计，得以只通过简单的 DDPM 就可以完成图像修复任务．但是，大多数的图像处理的任务，例如图像着色、图像超分辨等，都需要训练一个更为复杂的网络，把预期的最终输出图像也输入给网络．这类算法称为 Image-to-Image 算法，其中比较简单的算法包括图像超分辨的 SR3 算法，和可用于图像着色、修复等多种任务的 Palette 算法等．

在 Image-to-Image 的任务中，我们稍稍改变一下记号，把输入的图像记作 $\boldsymbol{x}$，而把扩散过程中各个带有噪声的图像记作 $\boldsymbol{y}_t$．也就是说，我们希望在给定了 $\boldsymbol{x}$ 后，算法可以通过一个高斯噪声的采样 $\boldsymbol{y}_T$ 进行逆向扩散 $\boldsymbol{y}_0 \leftarrow \cdots, \leftarrow \boldsymbol{y}_T$，得到输出 $\boldsymbol{y}_0$．例如，在图像着色任务中，$\boldsymbol{x}$ 就是输入的黑白图像，而生成的 $\boldsymbol{y}_0$ 则是对应的彩色图像．中间的反向扩散过程使用神经网络 $\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{y}_t, t)$ 来估计噪声． 这个网络的训练方式和 DDPM 并没有太大的区别，其损失函数如下，和 DDPM 相比只不过多了一个参数 $\boldsymbol{x}$ 而已．

SR3 和 Palette 原始论文的记号和 DDPM 又不一样，我们在这里还是和 DDPM 保持统一，除了 $\boldsymbol{y}$ 以外．

$$L = \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y}),\boldsymbol{\epsilon},t}\left[\left\|\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}, \underbrace{\sqrt{\bar{\alpha}_t}\boldsymbol{y}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}}_{\boldsymbol{y}_t}, t) - \boldsymbol{\epsilon}\right\|^2\right] \tag{5.10}$$

需要指出的是，我们训练用的数据集是与输出的图像 $\boldsymbol{y}_0$ 一致的，而训练时的 $\boldsymbol{x}$ 是由 $\boldsymbol{y}_0$ 得到的．例如对于图像着色任务，训练数据集应该是彩色图像 $\boldsymbol{y}_0$，而 $\boldsymbol{x}$ 则是由 $\boldsymbol{y}_0$ 转换而成的黑白图像．而在使用训练好的模型时，我们喂给模型的是一个黑白图像 $\boldsymbol{x}$，模型生会据此成一个彩色图像 $\boldsymbol{y}_0$．

5.4　一些技术细节

在最后一节，我们来讨论实现扩散模型的几个技术细节．

5.4.1　度量函数

在图像生成任务中，我们经常需要计算两个图像之间的差异作为损失函数的一部分．由于我们常常假设某些分布满足高斯分布，因此最终得到的损失函数往往是 L^2 损失，又称为 MSE 损失．不过，以 L^2 作为重构损失生成的图像经常会比较模糊．因此，重构损

失也常常会被直接替换成其他类型的损失函数．例如，L^1 损失，又称为 MAE 损失，在一些情况下可以解决图像模糊的问题．不过，和 L^1 损失相比，更常用的一种损失函数称为 **SSIM** (Structural Similarity Index Measure) 损失．它考虑了图像的亮度、对比度和结构，因此能更好地衡量图像之间的差异．SSIM 定义为

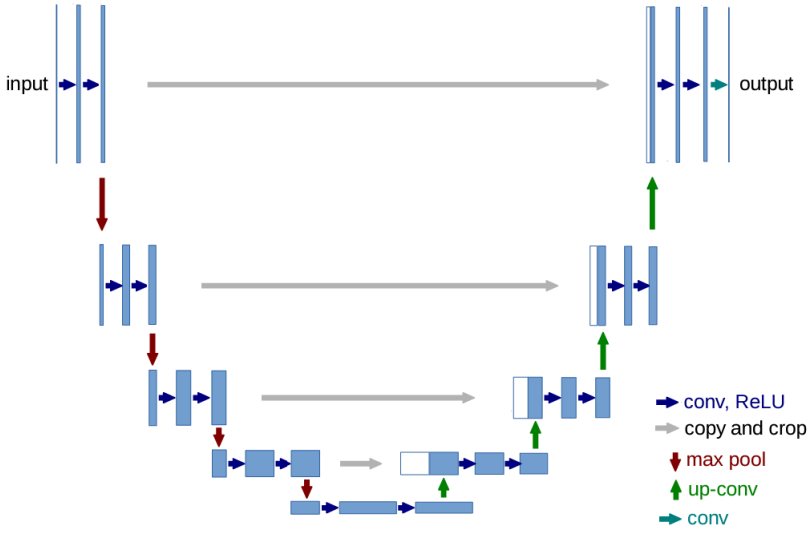
$$\mathrm{SSIM}(\boldsymbol{x}, \boldsymbol{y}) = \frac{(2\mu_{\boldsymbol{x}}\mu_{\boldsymbol{y}} + c_1)(2\sigma_{\boldsymbol{xy}} + c_2)}{(\mu_{\boldsymbol{x}}^2 + \mu_{\boldsymbol{y}}^2 + c_1)(\sigma_{\boldsymbol{x}}^2 + \sigma_{\boldsymbol{y}}^2 + c_2)}$$

以上损失函数都是以图像的像素值为基础的．不过，图像生成任务的最终目的往往是生成在人看起来真实或满足条件的图像．因此，人们建立了一类以人的主观感知差异未基础的损失函数，称为 **感知损失** (perceptual loss)．其中最常用的称为 **LPIPS** (Learned Perceptual Image Patch Similarity) 损失．LPIPS 是一个基于人类被试对图像相似度的判断所训练的一个神经网络，因此它可以更好地代表图像在人类看起来的差异大小．以 LPIPS 为损失函数重构出来的图像在人来看起来差异也会更小．

不过需要指出的是，这几种高级损失函数度量的是有意义的图像之间的差异．例如在 VAE 或一致性模型中，我们可以使用这几种高级的损失函数．但是，在 DDPM 和 DDIM 中，网络的输出是预测的噪声．计算噪声之间的差异仍然应该使用 L^2 损失．

5.4.2 网络架构

扩散模型里需要使用输入和输出是相同维度的神经网络．在发展的早期，计算机视觉里最主流的网络架构是 **卷积神经网络** (Convolutional Neural Network, **CNN**)．而扩散模型使用最多的 CNN 架构称为 **U-Net**，它由对称的卷积层和反卷积层构成，并在中间添加了跳跃连接 (skip connections)，如下图所示．通常来说，这里的卷积层都会加入残差连接．

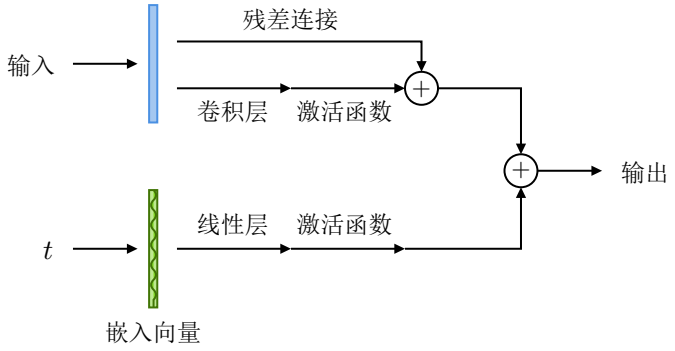


这里还有几个需要讨论的地方．首先，扩散模型中使用的网络大多都是形如 $\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}, t)$ 的网络，它需要时间 t 作为输入． 为了将时间 t 的信息嵌入网络中，U-Net 使用了类似于 Transformer 的位置嵌入 (position embedding) 的方式，首先将 t 转化为如下的正弦嵌入向量

有点网络输入的是噪声的方差 σ，但它和 t 本质上代表的都是噪声强度．

$$\mathrm{TE}_i(t) = \begin{cases} \sin\left(\dfrac{t}{10000^{i/d}}\right), & i \text{ 是偶数} \\ \cos\left(\dfrac{t}{10000^{i/d}}\right), & i \text{ 是奇数} \end{cases}$$

其中 d 是嵌入维度．之后，这个时间嵌入会送入一个神经网络中，这个网络的输出会加到 U-Net 的一些层的输出，作为下一层的输入．这样，加入了时间嵌入的的每一个模块如下图所示．

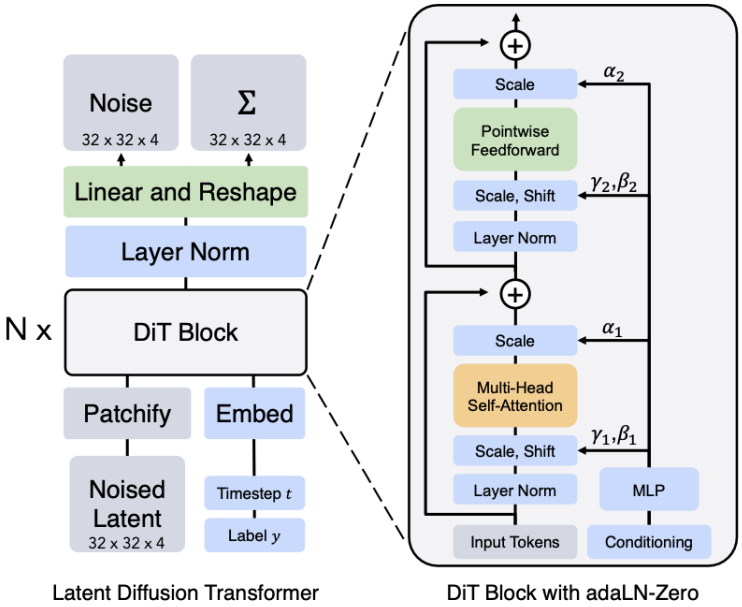


第二，通常网络架构都会加入 **注意力模块** (attention block)，以更好地提取图像中的结构信息．注意力模块通常被置于卷积层之后，并且只出现在分辨率较低的部分（U-Net 结构图中靠下的部分），以减少计算需求．实验表明，注意力模块对于扩散模型性能的提升非常重要．

对于条件扩散模型，有不同的方式将条件信息传入神经网络中．例如在普通的无分类器网络中，条件变量作为一个向量，可以以与时间类似的方式嵌入进网络中．而对于 Image-to-Image 网络，条件 $\boldsymbol{x}$ 可以和输入 $\boldsymbol{y}_t$ 拼接在一起输入网络中．

而近些年，起源于自然语言处理的 Transformer 架构逐渐进入计算机视觉领域，有取代 CNN 的趋势． 计算机视觉里最为主流的 Transformer 架构称为 **Vision Transformer** (**ViT**)．而 ViT 架构应用于扩散模型即为 **Diffusion Transformer** (**DiT**)．其基本结构如下图所示．

实际上，上面的注意力模块就来自 Transformer 架构．



在 DiT 架构中，输入的图像先被分割为小**块** (patches)， 这些小块会被一个线性层转化为一维向量，按顺序排成一个向量序列．接下来，这个向量序列和位置嵌入向量被输入含有自注意力的 DiT 模块中．DiT 模块与 Transformer 里的自注意力模块加 MLP 模块

这一步是将图像转化为 Transformer 所擅长的序列数据，每一个块类似于 NLP 里的一个词元 (token)．

非常类似，只是多了一个条件输入而已． DiT 模块的输出重新被线性转化为二维的块后，拼在一起，就得到了输出的图像．

本笔记的主要参考包括：

arXiv 文章：

1. DDPM 原始论文 https://arxiv.org/abs/2006.11239
2. NCSN 原始论文 https://arxiv.org/abs/1907.05600
3. DDIM 原始论文 https://arxiv.org/abs/2010.02502
4. RePaint 原始论文 https://arxiv.org/abs/2201.09865
5. SR3 原始论文 https://arxiv.org/abs/2104.07636
6. Palette 原始论文 https://arxiv.org/abs/2111.05826
7. 扩散模型的一个教程 https://arxiv.org/abs/2403.18103
8. U-Net 的示意图来自其原始论文 https://arxiv.org/abs/1505.04597
9. DiT 原始论文 https://arxiv.org/abs/2212.09748

个人博客：

1. Lilian Weng 的博文 https://lilianweng.github.io/
2. 苏剑林的博文 https://kexue.fm/

其他：

1. RePaint 的一张示意图改编自 UCF 的一个小组作业 https://www.crcv.ucf.edu/wp-content/uploads/2018/11/Paper-16-RePaint-Group-6.pdf
2. LabML 的 U-Net 代码讲解 https://nn.labml.ai/diffusion/ddpm/unet.html

本笔记的大部分图像来自 AFHQ 数据集