Advanced machine learning and data analysis for the physical sciences

Morten Hjorth-Jensen[1]

Department of Physics and Center for Computing in Science Education,
University of Oslo, Norway[1]

May 8, 2025

Plans for the week of May 5-9, 2025

Deep generative models

1. Mathematics of diffusion models and selected examples

Readings

Reading on diffusion models

1. A central paper is the one by Sohl-Dickstein et al, Deep Unsupervised Learning using Nonequilibrium Thermodynamics, https://arxiv.org/abs/1503.03585
2. Calvin Luo at https://arxiv.org/abs/2208.11970
3. See also Diederik P. Kingma, Tim Salimans, Ben Poole, Jonathan Ho, Variational Diffusion Models, https://arxiv.org/abs/2107.00630

Diffusion models, basics

Diffusion models are inspired by non-equilibrium thermodynamics. They define a Markov chain of diffusion steps to slowly add random noise to data and then learn to reverse the diffusion process to construct desired data samples from the noise. Unlike VAE or flow models, diffusion models are learned with a fixed procedure and the latent variable has high dimensionality (same as the original data).

Problems with probabilistic models

Historically, probabilistic models suffer from a tradeoff between two conflicting objectives: *tractability* and *flexibility*. Models that are *tractable* can be analytically evaluated and easily fit to data (e.g. a Gaussian or Laplace). However, these models are unable to aptly describe structure in rich datasets. On the other hand, models that are *flexible* can be molded to fit structure in arbitrary data. For example, we can define models in terms of any (non-negative) function $\phi(\boldsymbol{x})$ yielding the flexible distribution $p(\boldsymbol{x}) = \frac{\phi(\boldsymbol{x})}{Z}$, where Z is a normalization constant. However, computing this normalization constant is generally intractable. Evaluating, training, or drawing samples from such flexible models typically requires a very expensive Monte Carlo process.

Diffusion models

Diffusion models have several interesting features

- extreme flexibility in model structure,
- exact sampling,
- easy multiplication with other distributions, e.g. in order to compute a posterior, and
- the model log likelihood, and the probability of individual states, to be cheaply evaluated.

Original idea

In the original formulation, one uses a Markov chain to gradually convert one distribution into another, an idea used in non-equilibrium statistical physics and sequential Monte Carlo. Diffusion models build a generative Markov chain which converts a simple known distribution (e.g. a Gaussian) into a target (data) distribution using a diffusion process. Rather than use this Markov chain to approximately evaluate a model which has been otherwise defined, one can explicitly define the probabilistic model as the endpoint of the Markov chain. Since each step in the diffusion chain has an analytically evaluable probability, the full chain can also be analytically evaluated.

Diffusion learning

Learning in this framework involves estimating small perturbations to a diffusion process. Estimating small, analytically tractable, perturbations is more tractable than explicitly describing the full distribution with a single, non-analytically-normalizable, potential function. Furthermore, since a diffusion process exists for any smooth target distribution, this method can capture data distributions of arbitrary form.

Mathematics of diffusion models

Let us go back our discussions of the variational autoencoders from the lecture of April 24, see https://github.com/CompPhysics/AdvancedMachineLearning/blob/main/doc/pub/week14/ipynb/week14.ipynb. As a first attempt at understanding diffusion models, we can think of these as stacked VAEs, or better, recursive VAEs.

Let us try to see why. As an intermediate step, we consider so-called hierarchical VAEs, which can be seen as a generalization of VAEs that include multiple hierarchies of latent spaces.

Note: Many of the derivations and figures here are inspired and borrowed from the excellent exposition of diffusion models by Calvin Luo at https://arxiv.org/abs/2208.11970.

Chains of VAEs

Markovian VAEs represent a generative process where we use Markov chain to build a hierarchy of VAEs.
Each transition down the hierarchy is Markovian, where we decode each latent set of variables $\boldsymbol{h}_t$ in terms of the previous latent variable $\boldsymbol{h}_{t-1}$. Intuitively, and visually, this can be seen as simply stacking VAEs on top of each other (see figure next slide).
One can think of such a model as a recursive VAE.

Mathematical representation

Mathematically, we represent the joint distribution and the posterior of a Markovian VAE as

$$p(\boldsymbol{x}, \boldsymbol{h}_{1:T}) = p(\boldsymbol{h}_T) p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{h}_1) \prod_{t=2}^{T} p_{\boldsymbol{\theta}}(\boldsymbol{h}_{t-1}|\boldsymbol{h}_t)$$

$$q_{\boldsymbol{\phi}}(\boldsymbol{h}_{1:T}|\boldsymbol{x}) = q_{\boldsymbol{\phi}}(\boldsymbol{h}_1|\boldsymbol{x}) \prod_{t=2}^{T} q_{\boldsymbol{\phi}}(\boldsymbol{h}_t|\boldsymbol{h}_{t-1})$$

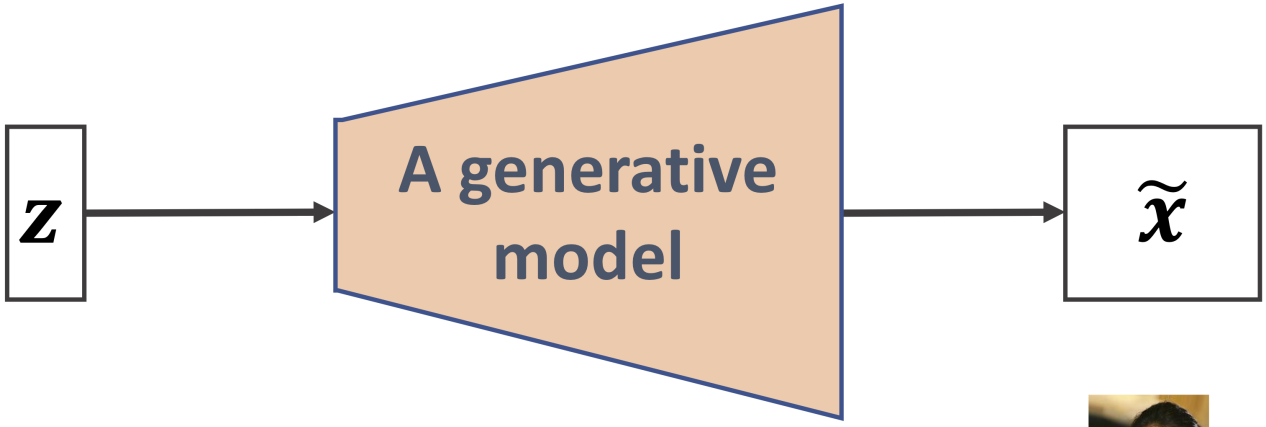
Back to the marginal probability

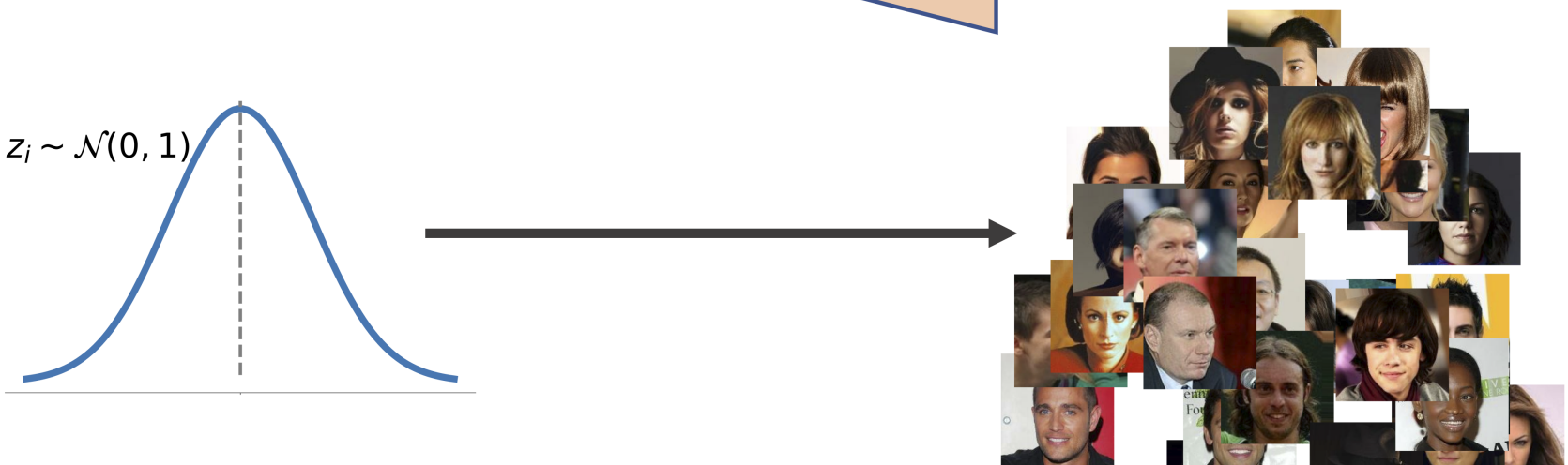
We can then define the marginal probability we want to optimize as

$$\begin{aligned}
\log p(\boldsymbol{x}) &= \log \int p(\boldsymbol{x}, \boldsymbol{h}_{1:T}) d\boldsymbol{h}_{1:T} \\
&= \log \int \frac{p(\boldsymbol{x}, \boldsymbol{h}_{1:T}) q_\phi(\boldsymbol{h}_{1:T}|\boldsymbol{x})}{q_\phi(\boldsymbol{h}_{1:T}|\boldsymbol{x})} d\boldsymbol{h}_{1:T} && \text{(Multiply by 1 = } \frac{q_\phi(\boldsymbol{h}_{1:T}|}{q_\phi(\boldsymbol{h}_{1:T}|} \\
&= \log \mathbb{E}_{q_\phi(\boldsymbol{h}_{1:T}|\boldsymbol{x})} \left[\frac{p(\boldsymbol{x}, \boldsymbol{h}_{1:T})}{q_\phi(\boldsymbol{h}_{1:T}|\boldsymbol{x})} \right] && \text{(Definition of Expectation)} \\
&\geq \mathbb{E}_{q_\phi(\boldsymbol{h}_{1:T}|\boldsymbol{x})} \left[\log \frac{p(\boldsymbol{x}, \boldsymbol{h}_{1:T})}{q_\phi(\boldsymbol{h}_{1:T}|\boldsymbol{x})} \right] && \text{(Discussed last week)}
\end{aligned}$$

Diffusion models for hierarchical VAE, from https://arxiv.org/abs/2208.11970

A Markovian hierarchical Variational Autoencoder with T hierarchical latents. The generative process is modeled as a Markov chain, where each latent $\boldsymbol{h}_t$ is generated only from the previous latent $\boldsymbol{h}_{t+1}$. Here $\boldsymbol{z}$ is our latent variable $\boldsymbol{h}$.





Equation for the Markovian hierarchical VAE

We obtain then

$$\mathbb{E}_{q_\phi(\boldsymbol{h}_{1:T}|\boldsymbol{x})}\left[\log \frac{p(\boldsymbol{x},\boldsymbol{h}_{1:T})}{q_\phi(\boldsymbol{h}_{1:T}|\boldsymbol{x})}\right] = \mathbb{E}_{q_\phi(\boldsymbol{h}_{1:T}|\boldsymbol{x})}\left[\log \frac{p(\boldsymbol{h}_T)p_\theta(\boldsymbol{x}|\boldsymbol{h}_1)\prod_{t=2}^{T} p_\theta(\boldsymbol{h}_{t-1}|\boldsymbol{h}_t)}{q_\phi(\boldsymbol{h}_1|\boldsymbol{x})\prod_{t=2}^{T} q_\phi(\boldsymbol{h}_t|\boldsymbol{h}_{t-1})}\right]$$

We will modify this equation when we discuss what are normally called Variational Diffusion Models.

Variational Diffusion Models

The easiest way to think of a Variational Diffusion Model (VDM) is as a Markovian Hierarchical Variational Autoencoder with three key restrictions:

1. The latent dimension is exactly equal to the data dimension
2. The structure of the latent encoder at each timestep is not learned; it is pre-defined as a linear Gaussian model. In other words, it is a Gaussian distribution centered around the output of the previous timestep
3. The Gaussian parameters of the latent encoders vary over time in such a way that the distribution of the latent at final timestep T is a standard Gaussian

The VDM posterior is

$$q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0) = \prod_{t=1}^{T} q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})$$

Second assumption

The distribution of each latent variable in the encoder is a Gaussian centered around its previous hierarchical latent. Here then, the structure of the encoder at each timestep t is not learned; it is fixed as a linear Gaussian model, where the mean and standard deviation can be set beforehand as hyperparameters, or learned as parameters.

Parameterizing Gaussian encoder

We parameterize the Gaussian encoder with mean $\boldsymbol{\mu}_t(\boldsymbol{x}_t) = \sqrt{\alpha_t}\boldsymbol{x}_{t-1}$, and variance $\boldsymbol{\Sigma}_t(\boldsymbol{x}_t) = (1 - \alpha_t)\mathbf{I}$, where the form of the coefficients are chosen such that the variance of the latent variables stay at a similar scale; in other words, the encoding process is variance-preserving.

Note that alternate Gaussian parameterizations are allowed, and lead to similar derivations. The main takeaway is that α_t is a (potentially learnable) coefficient that can vary with the hierarchical depth t, for flexibility.

Encoder transitions

Mathematically, the encoder transitions are defined as

$$q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}) = \mathcal{N}(\boldsymbol{x}_t; \sqrt{\alpha_t}\boldsymbol{x}_{t-1}, (1-\alpha_t)\mathbf{I})$$

Third assumption

From the third assumption, we know that α_t evolves over time according to a fixed or learnable schedule structured such that the distribution of the final latent $p(\boldsymbol{x}_T)$ is a standard Gaussian. We can then update the joint distribution of a Markovian VAE to write the joint distribution for a VDM as

$$p(\boldsymbol{x}_{0:T}) = p(\boldsymbol{x}_T) \prod_{t=1}^{T} p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$$

where,

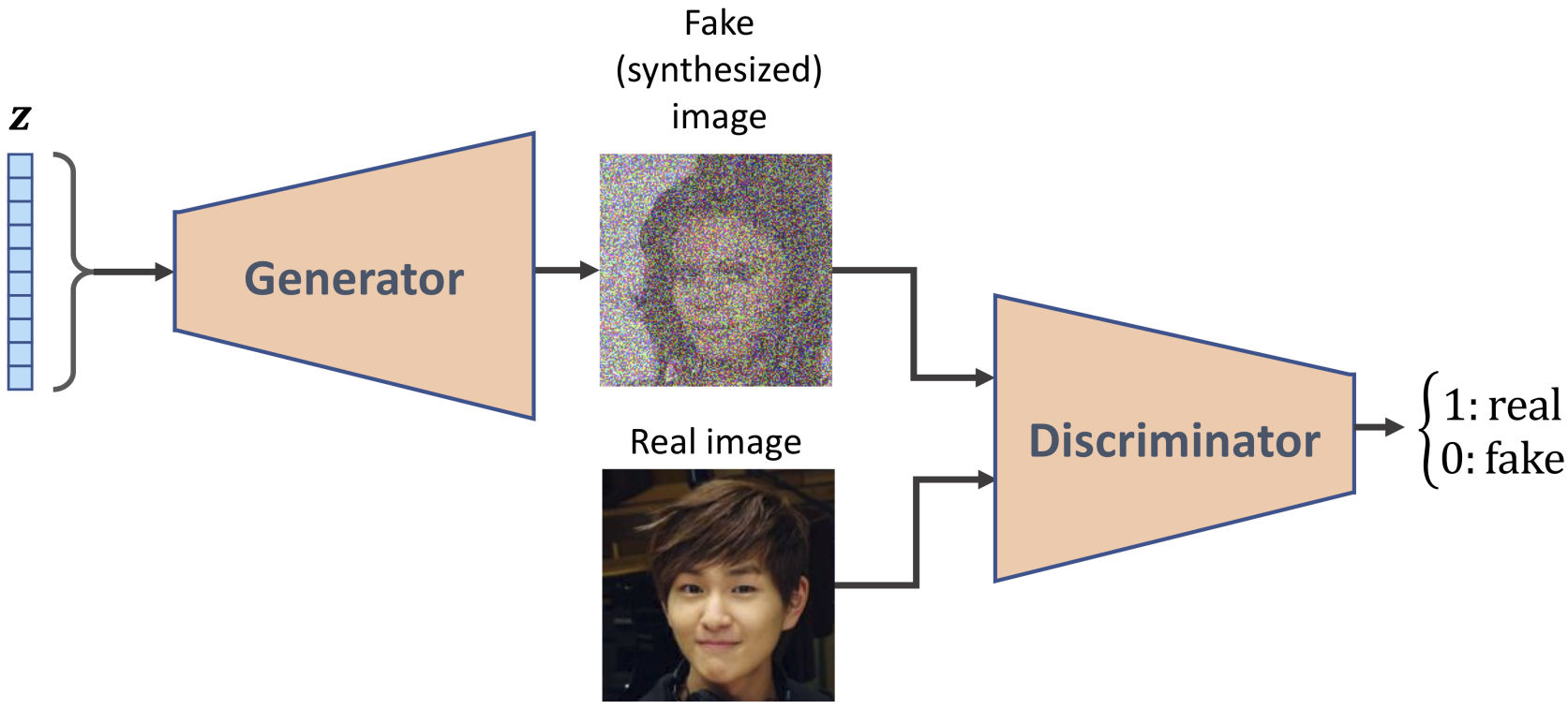
$$p(\boldsymbol{x}_T) = \mathcal{N}(\boldsymbol{x}_T; 0, \mathbf{I})$$

Noisification

Collectively, what this set of assumptions describes is a steady noisification of an image input over time. We progressively corrupt an image by adding Gaussian noise until eventually it becomes completely identical to pure Gaussian noise. See figure on next slide.

Diffusion models, from

https://arxiv.org/abs/2208.11970



Gaussian modeling

Note that our encoder distributions $q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})$ are no longer parameterized by ϕ, as they are completely modeled as Gaussians with defined mean and variance parameters at each timestep. Therefore, in a VDM, we are only interested in learning conditionals $p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$, so that we can simulate new data. After optimizing the VDM, the sampling procedure is as simple as sampling Gaussian noise from $p(\boldsymbol{x}_T)$ and iteratively running the denoising transitions $p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ for T steps to generate a novel $\boldsymbol{x}_0$.

Optimizing the variational diffusion model

$$
\begin{aligned}
\log p(\boldsymbol{x}) &= \log \int p(\boldsymbol{x}_{0:T}) d\boldsymbol{x}_{1:T} \\
&= \log \int \frac{p(\boldsymbol{x}_{0:T}) q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)} d\boldsymbol{x}_{1:T} \\
&= \log \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)} \left[\frac{p(\boldsymbol{x}_{0:T})}{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)} \right] \\
&\geq \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)} \left[\log \frac{p(\boldsymbol{x}_{0:T})}{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)} \right] \\
&= \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)} \left[\log \frac{p(\boldsymbol{x}_T) \prod_{t=1}^{T} p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)}{\prod_{t=1}^{T} q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})} \right] \\
&= \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)} \left[\log \frac{p(\boldsymbol{x}_T) p_\theta(\boldsymbol{x}_0|\boldsymbol{x}_1) \prod_{t=2}^{T} p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)}{q(\boldsymbol{x}_T|\boldsymbol{x}_{T-1}) \prod_{t=1}^{T-1} q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})} \right] \\
&= \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)} \left[\log \frac{p(\boldsymbol{x}_T) p_\theta(\boldsymbol{x}_0|\boldsymbol{x}_1) \prod_{t=1}^{T-1} p_\theta(\boldsymbol{x}_t|\boldsymbol{x}_{t+1})}{q(\boldsymbol{x}_T|\boldsymbol{x}_{T-1}) \prod_{t=1}^{T-1} q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})} \right] \\
&= \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)} \left[\log \frac{p(\boldsymbol{x}_T) p_\theta(\boldsymbol{x}_0|\boldsymbol{x}_1)}{q(\boldsymbol{x}_T|\boldsymbol{x}_{T-1})} \right] + \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)} \left[\log \prod_{t=1}^{T-1} p_\theta(\boldsymbol{x}_t|\boldsymbol{x}_{t+1}) \right] \\
&\quad - \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)} \left[\log \prod_{t=1}^{T-1} q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}) \right]
\end{aligned}
$$

Continues

$$\log p(\boldsymbol{x}) = \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}\left[\log \frac{p(\boldsymbol{x}_T)p_\theta(\boldsymbol{x}_0|\boldsymbol{x}_1)}{q(\boldsymbol{x}_T|\boldsymbol{x}_{T-1})}\right] + \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}\left[\log \prod_{t=1}^{T-1} \frac{p_\theta(\boldsymbol{x}_t|\boldsymbol{x}_t}{q(\boldsymbol{x}_t|\boldsymbol{x}_t}\right.$$

$$= \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}\left[\log p_\theta(\boldsymbol{x}_0|\boldsymbol{x}_1)\right] + \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}\left[\log \frac{p(\boldsymbol{x}_T)}{q(\boldsymbol{x}_T|\boldsymbol{x}_{T-1})}\right] + \mathbb{E}_{q(}$$

$$= \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}\left[\log p_\theta(\boldsymbol{x}_0|\boldsymbol{x}_1)\right] + \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}\left[\log \frac{p(\boldsymbol{x}_T)}{q(\boldsymbol{x}_T|\boldsymbol{x}_{T-1})}\right] + \sum_{t=1}^{T-1}$$

$$= \mathbb{E}_{q(\boldsymbol{x}_1|\boldsymbol{x}_0)}\left[\log p_\theta(\boldsymbol{x}_0|\boldsymbol{x}_1)\right] + \mathbb{E}_{q(\boldsymbol{x}_{T-1},\boldsymbol{x}_T|\boldsymbol{x}_0)}\left[\log \frac{p(\boldsymbol{x}_T)}{q(\boldsymbol{x}_T|\boldsymbol{x}_{T-1})}\right] + \sum_{t=}^{T}$$

Interpretations

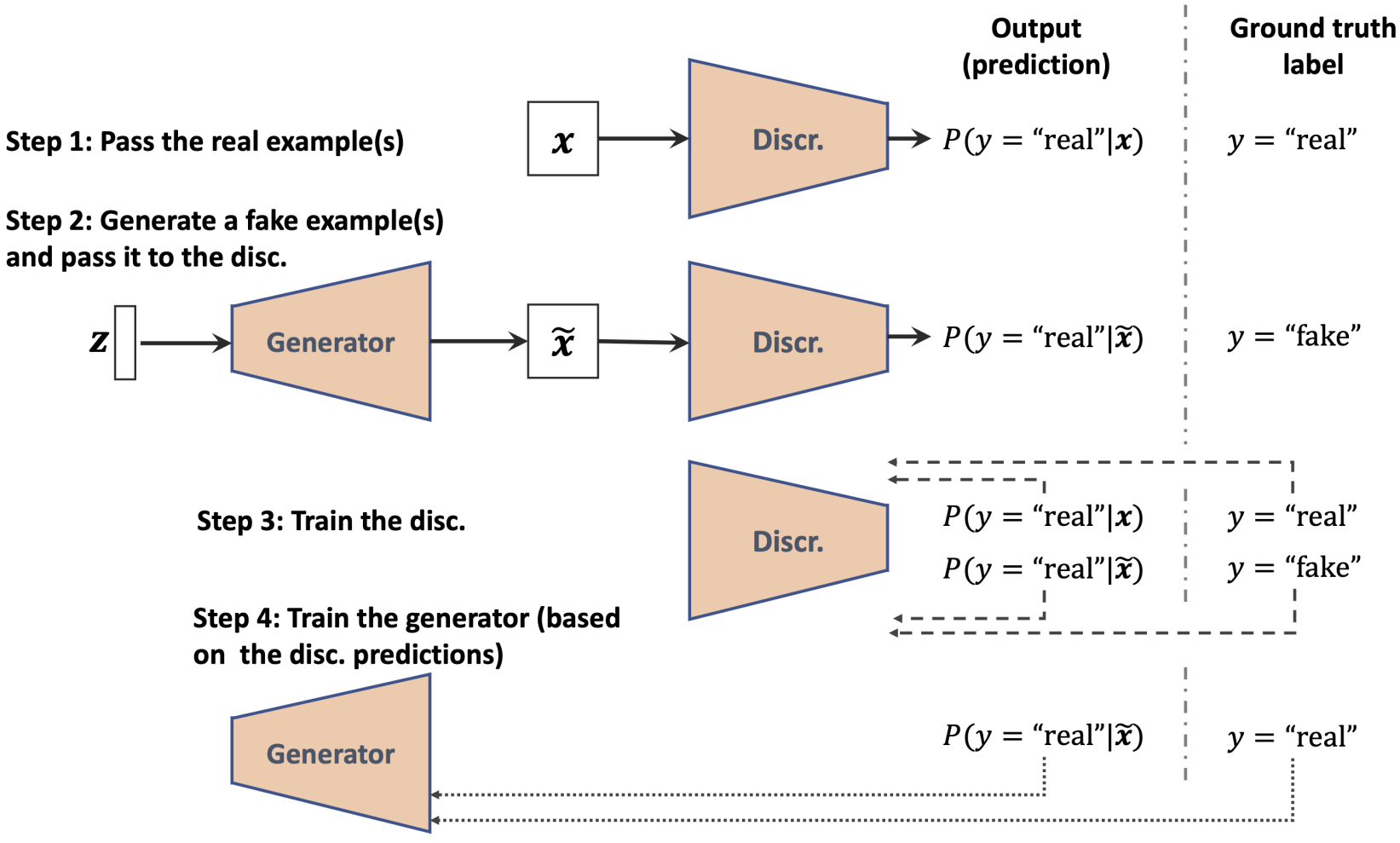
These equations can be interpreted as

- $\mathbb{E}_{q(\boldsymbol{x}_1|\boldsymbol{x}_0)}[\log p_\theta(\boldsymbol{x}_0|\boldsymbol{x}_1)]$ can be interpreted as a **reconstruction term**, predicting the log probability of the original data sample given the first-step latent. This term also appears in a vanilla VAE, and can be trained similarly.
- $\mathbb{E}_{q(\boldsymbol{x}_{T-1}|\boldsymbol{x}_0)}[D_{KL}(q(\boldsymbol{x}_T|\boldsymbol{x}_{T-1})||p(\boldsymbol{x}_T))]$ is a **prior matching term**; it is minimized when the final latent distribution matches the Gaussian prior. This term requires no optimization, as it has no trainable parameters; furthermore, as we have assumed a large enough T such that the final distribution is Gaussian, this term effectively becomes zero.

The last term

- $\mathbb{E}_{q(\boldsymbol{x}_{t-1},\boldsymbol{x}_{t+1}|\boldsymbol{x}_0)}\left[D_{KL}(q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})||p_{\boldsymbol{\theta}}(\boldsymbol{x}_t|\boldsymbol{x}_{t+1}))\right]$ is a *consistency term*; it endeavors to make the distribution at $\boldsymbol{x}_t$ consistent, from both forward and backward processes. That is, a denoising step from a noisier image should match the corresponding noising step from a cleaner image, for every intermediate timestep; this is reflected mathematically by the KL Divergence. This term is minimized when we train $p_\theta(\boldsymbol{x}_t|\boldsymbol{x}_{t+1})$ to match the Gaussian distribution $q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})$.

Diffusion models, part 2, from https://arxiv.org/abs/2208.11970



Optimization cost

The cost of optimizing a VDM is primarily dominated by the third term, since we must optimize over all timesteps t.
Under this derivation, all three terms are computed as expectations, and can therefore be approximated using Monte Carlo estimates. However, actually optimizing the ELBO using the terms we just derived might be suboptimal; because the consistency term is computed as an expectation over two random variables $\{\boldsymbol{x}_{t-1}, \boldsymbol{x}_{t+1}\}$ for every timestep, the variance of its Monte Carlo estimate could potentially be higher than a term that is estimated using only one random variable per timestep. As it is computed by summing up $T-1$ consistency terms, the final estimated value may have high variance for large T values.

PyTorch implementation of a Denoising Diffusion Probabilistic Model (DDPM) trained on the MNIST dataset

The code covers:

1. Model definition (a simple U-Net-style convolutional network)
2. Forward diffusion (adding noise over T timesteps)
3. Reverse denoising process
4. Training loop
5. Sampling from the trained model

This example is adapted from several open-source tutorials and implementations, demonstrating how to build a diffusion model from scratch in under 200 lines of PyTorch. I have borrowed extensively from

1. Jackson-Kang's PyTorch diffusion tutorial, see `https://github.com/Jackson-Kang/Pytorch-Diffusion-Model-Tutorial` and
2. awjuliani's PyTorch DDPM implementation, see `https://github.com/awjuliani/pytorch-diffusion`

Problem with diffusion models

Diffusion models gradually corrupt data by adding Gaussian noise over a sequence of timesteps and then learn to reverse this noising process with a neural network.
The corruption schedule is typically linear or cosine in variance.
During training, the network is optimized to predict the original noise added at each timestep, using a mean-squared error loss.
At inference, one starts from random noise and iteratively applies the learned denoising steps to generate new samples.

Imports and Utilities

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import math
```

Hyperparameters and schedules

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'

# Training settings
batch_size = 128
epochs     = 5
lr         = 2e-4
img_size   = 28
channels   = 1

# Diffusion hyperparameters
T = 300  # number of diffusion steps  [oai_citation:5‡Medium](https://
beta_start, beta_end = 1e-4, 0.02
betas = torch.linspace(beta_start, beta_end, T, device=device)  # line
alphas = 1. - betas
alphas_cumprod = torch.cumprod(alphas, dim=0)
```

Data Loading

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,)),
])

train_ds = datasets.MNIST('.', train=True, download=True, transform=tr
train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=Tru
```

Model definition

We present a lightweight U-Net inspired model for noise prediction:

```
class SimpleUNet(nn.Module):
    def __init__(self, c):
        super().__init__()
        self.enc1 = nn.Conv2d(c, 64, 3, padding=1)
        self.enc2 = nn.Conv2d(64, 128, 3, padding=1)
        self.dec1 = nn.ConvTranspose2d(128, 64, 3, padding=1)
        self.dec2 = nn.ConvTranspose2d(64, c, 3, padding=1)
        self.act  = nn.ReLU()
        # timestep embedding to condition on t
        self.time_mlp = nn.Sequential(
            nn.Linear(1, 64),
            nn.ReLU(),
            nn.Linear(64, 64),
        )

    def forward(self, x, t):
        # x: [B, C, H, W], t: [B]
        h = self.act(self.enc1(x))                          #  [oai_citat
        h = self.act(self.enc2(h))
        # add time embedding
        t = t.unsqueeze(-1)
        temb = self.time_mlp(t)                             #  [oai_citat
        temb = temb.view(-1, 64, 1, 1)
        h = h + temb
        h = self.act(self.dec1(h))
        return self.dec2(h)
```

Forward Diffusion $q(x_t|x_0)$

```
def q_sample(x0, t, noise=None):
    """Add noise to x0 at timestep t."""
    if noise is None:
        noise = torch.randn_like(x0)
    sqrt_acp = alphas_cumprod[t]**0.5
    sqrt_1macp = (1 - alphas_cumprod[t])**0.5
    return sqrt_acp.view(-1,1,1,1)*x0 + sqrt_1macp.view(-1,1,1,1)*nois
```

Cost/Loss function

```python
def diffusion_loss(model, x0):
    """Compute MSE between predicted noise and true noise."""
    B = x0.size(0)
    t = torch.randint(0, T, (B,), device=device).long()
    noise = torch.randn_like(x0)
    x_noisy = q_sample(x0, t, noise)
    pred_noise = model(x_noisy, t.float()/T)
    return F.mse_loss(pred_noise, noise)
```

Training loop

```
model = SimpleUNet(channels).to(device)
opt   = torch.optim.Adam(model.parameters(), lr=lr)

for epoch in range(epochs):
    total_loss = 0
    for x, _ in train_loader:
        x = x.to(device)
        loss = diffusion_loss(model, x)
        opt.zero_grad()
        loss.backward()
        opt.step()
        total_loss += loss.item()
    print(f"Epoch {epoch+1}/{epochs}, Loss: {total_loss/len(train_load
```

Sampling (Reverse Diffusion)

```
@torch.no_grad()
def p_sample_loop(model, shape):
    x = torch.randn(shape, device=device)
    for i in reversed(range(T)):
        t = torch.full((shape[0],), i, device=device).float()/T
        eps_pred = model(x, t)
        beta_t = betas[i]
        alpha_t = alphas[i]
        acp_t   = alphas_cumprod[i]
        coef1 = 1 / alpha_t.sqrt()
        coef2 = beta_t / ( (1 - acp_t).sqrt() )
        x = coef1*(x - coef2*eps_pred)
        if i > 0:
            z = torch.randn_like(x)
            sigma = beta_t.sqrt()
            x = x + sigma*z
    return x

# Generate samples
samples = p_sample_loop(model, (16, channels, img_size, img_size))
samples = samples.clamp(-1,1).cpu()
grid = torchvision.utils.make_grid(samples, nrow=4, normalize=True)
plt.figure(figsize=(5,5))
plt.imshow(grid.permute(1,2,0))
plt.axis('off')
```

More details

For more details and implementations, see Calvin Luo at
https://arxiv.org/abs/2208.11970