Diffusion Models

Dayta, Dominic Bagus

Mathematical Informatics Laboratory
Nara Institute of Science and Technology

31 July, 2025

Overview

Generative Models

Previous sections have demonstrated the power of generative latent models: we suppose the data space $\mathbf{x}$ arises from a latent space $\mathbf{z}$ with probability distribution $p(\mathbf{z})$.

Diffusion models, also called *denoising diffusion probabilistic models*, or DDPMs follows this same idea.

We take each training image x_i and to corrupt it using a multi-step noise process to transform it into a Gaussian sample z_i. A deep neural network is then trained to invert this process.

Generative Models

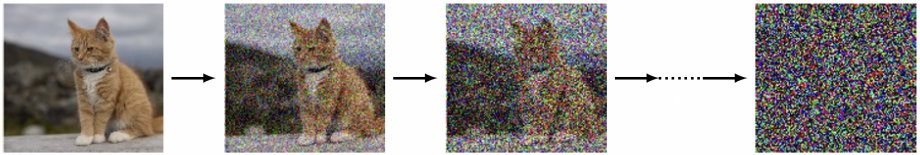


Figure 20.1 Illustration of the encoding process in a diffusion model showing an image $\mathbf{x}$ that is gradually corrupted with multiple stages of additive Gaussian noise giving a sequence of increasingly noisy images. After a large number T of steps the result is indistinguishable from a sample drawn from a Gaussian distribution. A deep neural network is then trained to reverse this process.

Diffusion models are technically not limited to image modeling, but provides a useful motivation.

Forward Encoder

Beginning with an original image $z_0 = x$, we can corrupt it with Gaussian noise at each pixel by applying the transformation

$$z_t = \sqrt{1-\beta_t} z_{t-1} + \sqrt{\beta_t}\epsilon_t$$

for $\epsilon_t \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. We choose β_t such that z_t is closer to z_{t-1} than ϵ_t. We can write the transformation in the form,

$$q(z_t|z_{t-1}) = \mathcal{N}(\sqrt{1-\beta_t} z_{t-1}, \beta_t \mathbf{I})$$

The values of the variance parameters $\beta_t \in (0, 1)$ are set by hand, such that the variance values increase through the chain according to a prescribed schedule such that $\beta_1 < \beta_2 < \cdots < \beta_T$

Forward Encoder

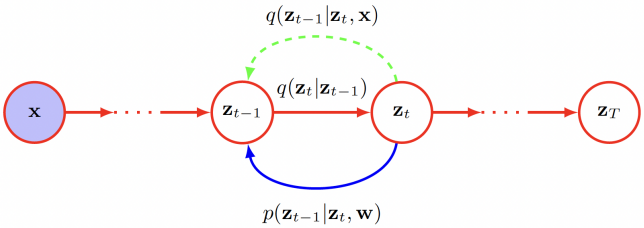


Figure 20.2 A diffusion process represented as a probabilistic graphical model. The original image $\mathbf{x}$ is shown by the shaded node, since it is an observed variable, whereas the noise-corrupted images $\mathbf{z}_1, \ldots, \mathbf{z}_T$ are considered to be latent variables. The noise process is defined by the forward distribution $q(\mathbf{z}_t|\mathbf{z}_{t-1})$ and can be viewed as an encoder. Our goal is to learn a model $p(\mathbf{z}_{t-1}|\mathbf{z}_t, \mathbf{w})$ that tries to reverse this noise process and which can be viewed as a decoder. As we will see later, the conditional distribution $q(\mathbf{z}_{t-1}|\mathbf{z}_t, \mathbf{x})$ plays an important role in defining the training procedure.

Forward Encoder

The joint distribution of the latent variables, conditioned on the observed data vector x, is given by

$$q(z_1, \ldots, z_t|x) = q(z_1|z_0) \prod_{\tau=1}^{t} q(z_\tau|z_{\tau-1})$$

Marginalizing over the intermediate variables $z_1, \ldots, z_{t-1}$ gives us the diffusion kernel.

$$q(z_T|z_0) = \mathcal{N}(\sqrt{\alpha_2} z_0, (1 - \alpha_2)\mathbf{I})$$

for $\alpha_t = \prod_{\tau=1}^{t}(1 - \beta_\tau)$

Diffusion Kernel

Let $\alpha_1 = 1 - \beta_1$, we know

$$z_1 = \sqrt{\alpha_1} z_0 + \sqrt{1 - \alpha_1} \epsilon_0$$
$$q(z_1|z_0) = \mathcal{N}(\sqrt{\alpha_1} z_0, (1 - \alpha_1)\mathbf{I})$$

Now, for z_2,

$$z_2 = \sqrt{1 - \beta_2} z_1 + \sqrt{\beta_2} \epsilon_1$$

Substituting the form we have for z_1

$$z_2 = \sqrt{1 - \beta_2} z_1 + \sqrt{\beta_2} \epsilon_1$$
$$z_2 = \sqrt{1 - \beta_2}(\sqrt{1 - \beta_1} z_0 + \sqrt{\beta_1} \epsilon_0) + \sqrt{\beta_2} \epsilon_1$$
$$z_2 = \sqrt{1 - \beta_2}\sqrt{1 - \beta_1} z_0 + \sqrt{1 - \beta_2}\sqrt{\beta_1} \epsilon_0 + \sqrt{\beta_2} \epsilon_1$$

Diffusion Kernel

The combined noise terms are in the form of the sum of two Gaussians, which we know to be Gaussian:

$$\begin{aligned} Var(\sqrt{1-\beta_2}\sqrt{\beta_1}\epsilon_0 + \sqrt{\beta_2}\epsilon_1) &= \beta_1(1-\beta_2) + \beta_2 \\ &= 1 - 1 + \beta_1 - \beta_1\beta_2 + \beta_2 \\ &= 1 - (1-\beta_1) - \beta_2(1-\beta_1) \\ &= 1 - (1-\beta_1)(1-\beta_2) \\ &= 1 - \alpha_2 \end{aligned}$$

Hence we have shown

$$q(z_2|z_0) = \mathcal{N}(\sqrt{\alpha_2}z_0, (1-\alpha_2)\mathbf{I})$$

The rest of the proof proceeds by induction.

Diffusion Kernel

This allows efficient stochastic gradient descent using randomly chosen intermediate terms in the Markov chain without having to run the whole chain. We can also write the kernel in the form

$$z_t = \sqrt{\alpha_t} z_0 + \sqrt{1 - \alpha_t} \epsilon_t$$

After many steps T this converges to the stationary distribution

$$q(z_T | z_0) = \mathcal{N}(\mathbf{0}, \mathbf{I})$$

as

$$\lim_{t \to \infty} \alpha_t = \lim_{t \to \infty} \prod_{\tau=1}^{t} (1 - \beta_\tau) = 0$$

Reverse Decoder

The forward encoder model is defined by a sequence of Gaussian conditional distributions $q(z_t|z_{t-1})$.

However, inverting this directly leads to a distribution $q(z_{t-1}|z_t)$ that is intractable, as it would require integrating over all possible values of the starting vector z_0.

Recall that the distribution of z_0 is the unknown data distribution $p(z_0 = x)$ that we wish to model.

Instead, we will learn an approximation to the reverse distribution by using a distribution $p(z_{t-1}|z_t|w)$ governed by a deep neural network.

Reverse Decoder

If $\beta_t << 1$, the distribution $q(z_{t-1}|z_t)$ will be approximately a Gaussian distribution over z_{t-1}.

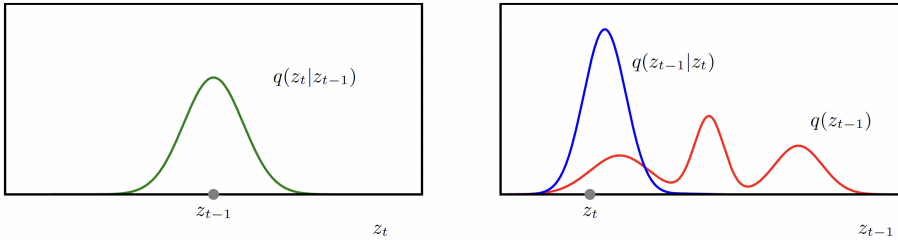


Figure 20.4 As in Figure 20.3 but in which the Gaussian distribution $q(z_t|z_{t-1})$ in the left-hand plot has a much smaller variance β_t. We see that the corresponding distribution $q(z_{t-1}|z_t)$ shown in blue on the right-hand plot is close to being Gaussian, with a similar variance to $q(z_t|z_{t-1})$.

Reverse Decoder

However, since the variances at each step are small, we must use a large number of steps.

This ensure that the distribution over the final latent variable z_T obtained from the forward noising process will still be close to a Gaussian.

This increases the cost of generating new samples. In practice, T may be several thousand.

Reverse Decoder

The overall reverse denoising process then takes the form of a Markov chain given by

$$p(z_0, z_1, \ldots, z_T|w) = p(z_T)\left\{\prod_{t=2}^{T} p(z_{t-1|z_t,w})\right\}p(z_0|z_1, w)$$

Here $p(z_T)$ is assumed to be the same as the distribution of $q(z_T)$ and hence is given by $\mathcal{N}(\mathbf{0}, \mathbf{I})$.

Once the model has been trained, we first sample from the simple Gaussian $p(z_T)$ and then again sequentially from each of the conditional distributions $p(z_{t-1}|z_t, w)$ in turn, finally sampling from $p(x|z_1, w)$ to obtain a sample $z_0 = x$ in the data space.

Training the Decoder

The full data log-likelihood involves integrating over all possible trajectories from noise to data point, and hence is intractable.

We can adopt a similar approach to that used with variational autoencoders and maximize a lower bound on the log likelihood, specifically the evidence lower bound (ELBO),

$$\mathcal{L}(w) = \int \ln \left\{ \frac{p(x, z, |w)}{q(z)} \right\} q(z) = \mathbb{E}\left[\ln \left\{ \frac{p(x, z, |w)}{q(z)} \right\} \right]$$

We chose $q(z)$ to be given by the fixed distribution $q(z_1, \ldots, z_T|x)$ defined by the Markov chain, and so the only adjustable parameters are those in the model $p(x, z_1, \ldots, z_T|w)$ for the reverse Markov chain.

Training the Decoder

Rewriting this with the chosen model,

$$\begin{aligned}\mathcal{L} &= \mathbb{E}\left[\ln \frac{p(z_T)\left\{\prod_{t=2}^{T} p(z_{t-1}|z_t, w)\right\} p(z_0|z_1, w)}{q(z_1|z_0)\prod_{t=2}^{T} q(z_t|z_{t-1}, z_0)}\right] \\ &= \mathbb{E}\left[\ln p(z_T) + \sum_{t=2}^{T} \ln \frac{p(z_{t-1}|z_t, w)}{q(z_t|z_{t-1}, z_0)} - \ln q(z_1|z_0) + \ln p(z_0|z_1, w)\right]\end{aligned}$$

Training the Decoder

This expectation is easy to estimate. First, we draw set of M samples of z_t. Let g be the complex function inside the expectation. Each jth draw z_t^* gives rise to g_j^*

$$g^* = \ln p(z_T^*) + \sum_{t=2}^{T} \ln \frac{p(z_{t-1}^*|z_t^*, w)}{q(z_t^*|z_{t-1}^*, z_0^*)} - \ln q(z_1^*|z_0) + \ln p(z_0^*|z_1^*, w)$$

The expectation $\mathbb{E}[g]$ then just becomes

$$\mathbb{E}[g] \approx \sum_{j=1}^{M} g_j^*$$

Training the Decoder

Require: Training data $\mathcal{D} = \{\mathbf{x}_n\}$, Noise schedule $\{\beta_1, \ldots, \beta_T\}$

for $t \in \{1, \ldots, T\}$ **do**
  $\alpha_t \leftarrow \prod_{\tau=1}^{t}(1 - \beta_\tau)$ ▷ Calculate alphas from betas
end for

repeat
  $\mathbf{x} \sim \mathcal{D}$ ▷ Sample a data point
  $t \sim \text{Uniform}(\{1, \ldots, T\})$ ▷ Sample a point along the Markov chain
  $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ ▷ Sample a noise vector
  $\mathbf{z}_t \leftarrow \sqrt{\alpha_t}\mathbf{x} + \sqrt{1 - \alpha_t}\boldsymbol{\epsilon}$ ▷ Evaluate noisy latent variable
  $\mathcal{L}(\mathbf{w}) \leftarrow \|\mathbf{g}(\mathbf{z}_t, \mathbf{w}, t) - \boldsymbol{\epsilon}\|^2$ ▷ Compute loss term
  Take optimizer step on $\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w})$
until converged

Training the Decoder

```
def q_sample(self, x_start, t, noise=None):
    """Forward process: add noise to an image."""
    if noise is None:
        noise = torch.randn_like(x_start)

    sqrt_alphas_cumprod_t = self.sqrt_alphas_cumprod[t].view(-1, 1, 1, 1)
    sqrt_one_minus_alphas_cumprod_t = self.sqrt_one_minus_alphas_cumprod[t].view(-1, 1, 1, 1)

    return sqrt_alphas_cumprod_t * x_start + sqrt_one_minus_alphas_cumprod_t * noise
```

Sampling from the Decoder

Require: Trained denoising network $\mathbf{g}(\mathbf{z}, \mathbf{w}, t)$, Noise schedule $\{\beta_1, \ldots, \beta_T\}$

$\mathbf{z}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ ▷ Sample from final latent space

for $t = T, \ldots, 2$ **do**

  $\alpha_t \leftarrow \prod_{\tau=1}^{t}(1 - \beta_\tau)$ ▷ Calculate alpha

  ▷ Evaluate network output

  $\boldsymbol{\mu}(\mathbf{z}_t, \mathbf{w}, t) \leftarrow \frac{1}{\sqrt{1-\beta_t}} \left\{ \mathbf{z}_t - \frac{\beta_t}{\sqrt{1-\alpha_t}} \mathbf{g}(\mathbf{z}_t, \mathbf{w}, t) \right\}$

  $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ ▷ Sample a noise vector

  $\mathbf{z}_{t-1} \leftarrow \boldsymbol{\mu}(\mathbf{z}_t, \mathbf{w}, t) + \sqrt{\beta_t}\boldsymbol{\epsilon}$ ▷ Add scaled noise

end for

$\mathbf{x} \leftarrow \frac{1}{\sqrt{1-\beta_1}} \left\{ \mathbf{z}_1 - \frac{\beta_1}{\sqrt{1-\alpha_1}} \mathbf{g}(\mathbf{z}_1, \mathbf{w}, 1) \right\}$ ▷ Final denoising step

Sampling from the Decoder

```
@torch.no_grad()
def sample(model, diffusion, n_images, img_size, channels=3, device="cpu"):
    """Samples new images from the diffusion model."""
    model.eval()

    img = torch.randn((n_images, channels, img_size, img_size), device=device)

    images = []
    for t in tqdm(reversed(range(0, diffusion.timesteps)), desc="Sampling",
                  total=diffusion.timesteps):
        t_tensor = torch.full((n_images,), t, device=device, dtype=torch.long)
        predicted_noise = model(img, t_tensor)

        alpha_t = diffusion.alphas[t]
        alpha_t_cumprod = diffusion.alphas_cumprod[t]
        beta_t = diffusion.betas[t]

        """More code below"""
```

Sampling from the Decoder

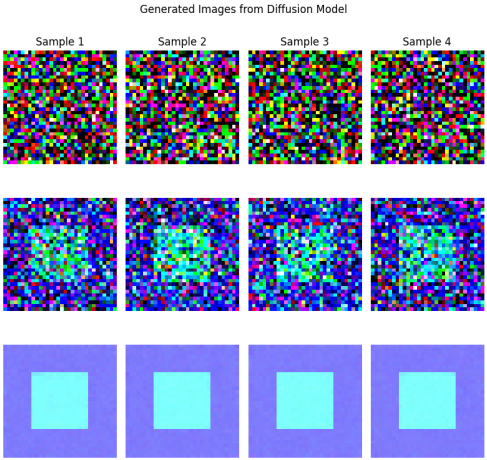
```
    """Continued from previous slide"""
    term1 = 1 / torch.sqrt(alpha_t)
    term2 = (beta_t / torch.sqrt(1 - alpha_t_cumprod)) * predicted_noise
    img = term1 * (img - term2)

    if t > 0:
        noise = torch.randn_like(img)
        img += torch.sqrt(beta_t) * noise

    if t % 50 == 0:
        images.append(img.cpu().numpy())

return img.cpu(), images
```

Sampling from the Decoder

Example of training a U-Net to sample images of a square.

The full code is available on my github:

@dominicdayta/diffuser



Score Matching

- At any step t, the noisy data distribution is $q(\mathbf{z}_t)$.
- The score function is defined as the gradient of the log-probability of this distribution with respect to the data:

$$\nabla_{\mathbf{z}_t} \log q(\mathbf{z}_t)$$

- he score is a vector field that points in the direction where the data density is increasing most rapidly.
- If we could learn this score function, we could use it to guide a reverse process, starting from random noise and moving towards regions of high data probability to generate a sample.

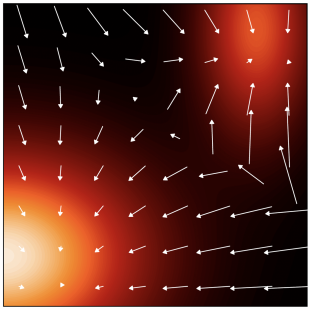


Figure 20.5 Illustration of the score function, showing a distribution in two dimensions comprising a mixture of Gaussians represented as a heat map and the corresponding score function defined by (20.42) plotted as vectors on a regular grid of x-values.

Score Matching

The core idea of score matching is to train a neural network, our score model $\mathbf{s}(\mathbf{z}_t, \mathbf{w}, t)$, to approximate the true (but unknown) score function.

We can define a loss function by minimizing the expected squared distance between our model's prediction and the true score:

$$\mathcal{L}_{SM}(\mathbf{w}) = \mathbb{E}_{q(\mathbf{z}_t)} \left[\|\mathbf{s}(\mathbf{z}_t, \mathbf{w}, t) - \nabla_{\mathbf{z}_t} \log q(\mathbf{z}_t)\|^2 \right] \tag{1}$$

- This objective directly trains our model to learn the gradient field of the data distribution.
- However, this formulation is still intractable because the target $\nabla_{\mathbf{z}_t} \log q(\mathbf{z}_t)$ depends on the unknown distribution $q(\mathbf{z}_t)$.

Score Matching

The solution is Denoising Score Matching.

The score of the conditional distribution $q(\mathbf{z}_t|\mathbf{x})$, which we can define, is directly related to the noise ϵ that was added:

$$\nabla_{\mathbf{z}_t} \log q(\mathbf{z}_t|\mathbf{x}) = -\frac{\mathbf{z}_t - \sqrt{\alpha_t}\mathbf{x}}{1-\alpha_t} = -\frac{\epsilon}{\sqrt{1-\alpha_t}}$$

Score Matching

It can be shown that minimizing the denoising objective is equivalent to minimizing the score matching objective.

Let our score model be defined by our noise-prediction network $\mathbf{g}(\mathbf{z}_t, \mathbf{w}, t)$:

$$\mathbf{s}(\mathbf{z}_t, \mathbf{w}, t) := -\frac{\mathbf{g}(\mathbf{z}_t, \mathbf{w}, t)}{\sqrt{1-\alpha_t}}$$

The denoising loss from Algorithm 20.1 is:

$$\mathcal{L}(\mathbf{w}) = \mathbb{E}_{t,\mathbf{x},\boldsymbol{\epsilon}}\left[\|\mathbf{g}(\mathbf{z}_t, \mathbf{w}, t) - \boldsymbol{\epsilon}\|^2\right]$$

Training a network $\mathbf{g}$ to predict the noise $\boldsymbol{\epsilon}$ is mathematically equivalent to training a network $\mathbf{s}$ to predict the score $\nabla_{\mathbf{z}_t} \log q(\mathbf{z}_t)$.

Guided Diffusion

The diffusion models we have seen so far are unconditional generators: they learn the entire distribution $p(\mathbf{x})$. In consequence, they can generate high-quality samples that look like the training data.

However, we have no control over the output. We cannot ask the model to generate an image of a specific class or with specific attributes.

Instead, we want to steer the generation process to sample from a conditional distribution $p(\mathbf{x}|\mathbf{y})$, where $\mathbf{y}$ is some conditioning information, such as:

- A class label (e.g., "cat", "dog", "car").
- A text description (e.g., "a photo of an astronaut riding a horse").
- Another image (for image-to-image translation).

Guided Diffusion

The core idea is to guide the reverse diffusion process towards a desired class $\mathbf{y}$.

1. We start with our standard unconditional diffusion model that predicts the score $\nabla_{\mathbf{z}_t} \log q(\mathbf{z}_t)$.
2. We separately train a classifier $p(\mathbf{y}|\mathbf{z}_t)$ that learns to predict the class label $\mathbf{y}$ from a noisy image $\mathbf{z}_t$.

Guided Diffusion

Using Bayes' theorem, the score of the desired *conditional* distribution is:

$$\underbrace{\nabla_{\mathbf{z}_t} \log q(\mathbf{z}_t|\mathbf{y})}_{\text{Guided Score}} = \underbrace{\nabla_{\mathbf{z}_t} \log q(\mathbf{z}_t)}_{\text{Unconditional Score}} + \underbrace{\nabla_{\mathbf{z}_t} \log p(\mathbf{y}|\mathbf{z}_t)}_{\text{Classifier Gradient}}$$

At each step t, we modify the mean of the reverse step. We nudge it in the direction of the classifier's gradient. A guidance scale s controls the strength of this effect.

This approach requires training and maintaining a separate classifier model on noisy images, which can be complex and unstable.

Guided Diffusion

A more elegant and powerful solution is to make the diffusion model itself handle the conditioning.

- The U-Net model is now conditioned on the class label $\mathbf{y}$: $\mathbf{g}(\mathbf{z}_t, \mathbf{w}, t, \mathbf{y})$.
- During training, we randomly replace the true label $\mathbf{y}$ with a null label $\varnothing$ (e.g., 10-20% of the time).
- This forces the same model to learn both the *conditional* prediction $\mathbf{g}(\ldots, \mathbf{y})$ and the *unconditional* prediction $\mathbf{g}(\ldots, \varnothing)$.

Guided Diffusion

At each sampling step, we compute both predictions and extrapolate from the unconditional towards the conditional estimate:

$$\hat{\epsilon} = \underbrace{\epsilon_\theta(\mathbf{z}_t, \varnothing)}_{\text{Unconditional}} + s \cdot \Big(\underbrace{\epsilon_\theta(\mathbf{z}_t, \mathbf{y})}_{\text{Conditional}} - \underbrace{\epsilon_\theta(\mathbf{z}_t, \varnothing)}_{\text{Unconditional}} \Big)$$

The term in parentheses is the "guidance direction".

The guidance scale $s > 1$ pushes the prediction further in this direction, improving sample quality and adherence to the condition $\mathbf{y}$.

Classifier-free guidance is simpler to implement, more stable to train, and generally produces higher-quality results. It is the standard for modern diffusion models.

Guided Diffusion

Figure 20.7 Illustration of classifier-free guidance of diffusion models, generated from a model called GLIDE using the conditioning text *A stained glass window of a panda eating bamboo*. Examples on the left were generated with $\lambda = 0$ (no guidance, just the plain conditional model) whereas examples on the right were generated with $\lambda = 3$. [From Nichol *et al.* (2021) with permission.]