DDPM

DDPM

Annotated-diffusion

A (denoising) diffusion model isn't that complex if you compare it to other generative models such as Normalizing Flows, GANs or VAEs: they all convert noise from some simple distribution to a data sample. This is also the case here where **a neural network learns to gradually denoise data** starting from pure noise.

In a bit more detail for images, the set-up consists of 2 processes:

- a fixed (or predefined) forward diffusion process q of our choosing, that gradually adds Gaussian noise to an image, until you end up with pure noise
- a learned reverse denoising diffusion process p_θ, where a neural network is trained to gradually denoise an image starting from pure noise, until you end up with an actual image.

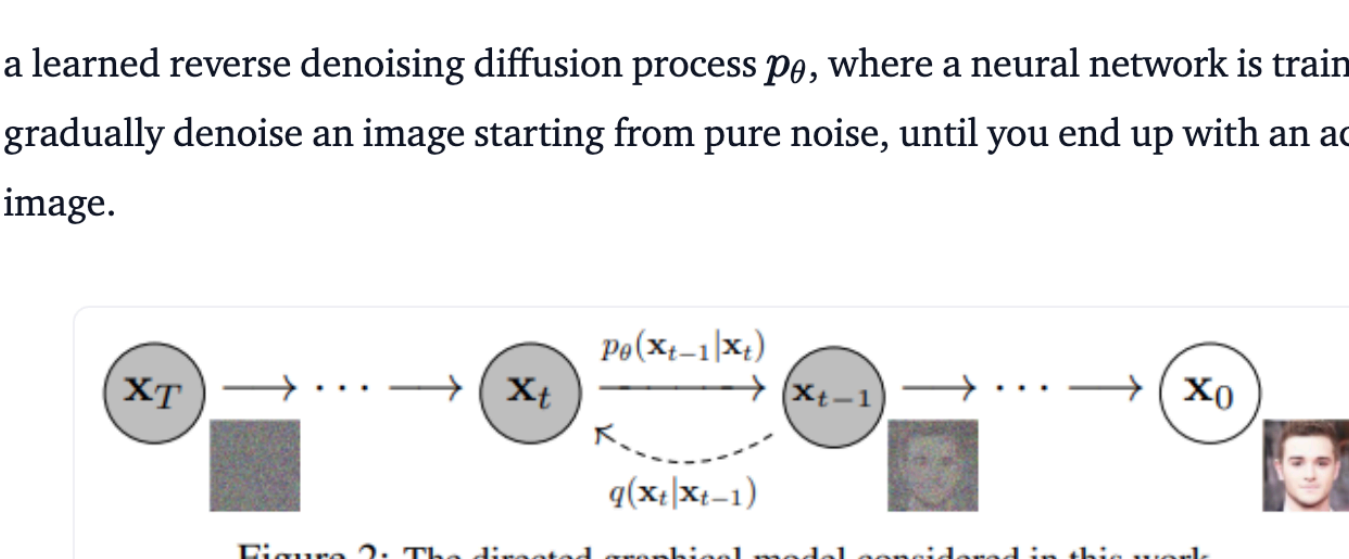

Figure 2: The directed graphical model considered in this work.

Both the forward and reverse process indexed by t happen for some number of finite time steps T (the DDPM authors use $T = 1000$). You start with $t = 0$ where you sample a real image $\mathbf{x}_0$ from your data distribution (let's say an image of a cat from ImageNet), and the forward process samples some noise from a Gaussian distribution at each time step t, which is added to the image of the previous time step. Given a sufficiently large T and a well behaved schedule for adding noise at each time step, you end up with what is called an isotropic Gaussian distribution at $t = T$ via a gradual process.

In more mathematical form

Let's write this down more formally, as ultimately we need a tractable loss function which our neural network needs to optimize.

Let $q(\mathbf{x}_0)$ be the real data distribution, say of "real images". We can sample from this distribution to get an image, $\mathbf{x}_0 \sim q(\mathbf{x}_0)$. We define the forward diffusion process $q(\mathbf{x}_t|\mathbf{x}_{t-1})$ which adds Gaussian noise at each time step t, according to a known variance schedule $0 < \beta_1 < \beta_2 < ... < \beta_T < 1$ as

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

Recall that a normal distribution (also called Gaussian distribution) is defined by 2 parameters: a mean μ and a variance $\sigma^2 \geq 0$. Basically, each new (slightly noisier) image at time step t is drawn from a **conditional Gaussian distribution** with $\mu_t = \sqrt{1-\beta_t}\mathbf{x}_{t-1}$ and $\sigma_t^2 = \beta_t$, which we can do by sampling $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and then setting $\mathbf{x}_t = \sqrt{1-\beta_t}\mathbf{x}_{t-1} + \sqrt{\beta_t}\epsilon$.

Note that the β_t aren't constant at each time step t (hence the subscript) --- in fact one defines a so-called "**variance schedule**", which can be linear, quadratic, cosine, etc. as we will see further (a bit like a learning rate schedule).

So starting from $\mathbf{x}_0$, we end up with $\mathbf{x}_1, ..., \mathbf{x}_t, ..., \mathbf{x}_T$, where $\mathbf{x}_T$ is pure Gaussian noise if we set the schedule appropriately.

Now, if we knew the conditional distribution $p(\mathbf{x}_{t-1}|\mathbf{x}_t)$, then we could run the process in reverse: by sampling some random Gaussian noise $\mathbf{x}_T$, and then gradually "denoise" it so that we end up with a sample from the real distribution $\mathbf{x}_0$.

However, we don't know $p(\mathbf{x}_{t-1}|\mathbf{x}_t)$. It's intractable since it requires knowing the distribution of all possible images in order to calculate this conditional probability. Hence, we're going to leverage a neural network to **approximate (learn) this conditional probability distribution**, let's call it $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$, with θ being the parameters of the neural network, updated by gradient descent.

Ok, so we need a neural network to represent a (conditional) probability distribution of the backward process. If we assume this reverse process is Gaussian as well, then recall that any Gaussian distribution is defined by 2 parameters:

- a mean parametrized by μ_θ;
- a variance parametrized by Σ_θ;

so we can parametrize the process as

$$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \mu_\theta(\mathbf{x}_t, t), \Sigma_\theta(\mathbf{x}_t, t))$$

where the mean and variance are also conditioned on the noise level t.

Hence, our neural network needs to learn/represent the mean and variance. However, the DDPM authors decided to **keep the variance fixed, and let the neural network only learn (represent) the mean μ_θ of this conditional probability distribution**. From the paper:

> *"First, we set $\Sigma_\theta(\mathbf{x}_t, t) = \sigma_t^2\mathbf{I}$ to untrained time dependent constants. Experimentally, both $\sigma_t^2 = \beta_t$ and $\sigma_t^2 = \tilde{\beta}_t$ (see paper) had similar results."*

This was then later improved in the Improved diffusion models paper, where a neural network also learns the variance of this backwards process, besides the mean.

So we continue, assuming that our neural network only needs to learn/represent the mean of this conditional probability distribution.

Defining an objective function (by reparametrizing the mean)

To derive an objective function to learn the mean of the backward process, the authors observe that the combination of q and p_θ can be seen as a variational auto-encoder (VAE) (Kingma et al., 2013). Hence, the **variational lower bound** (also called ELBO) can be used to minimize the negative log-likelihood with respect to ground truth data sample $\mathbf{x}_0$ (we refer to the VAE paper for details regarding ELBO). It turns out that the ELBO for this process is a sum of losses at each time step t, $L = L_0 + L_1 + ... + L_T$. By construction of the forward q process and backward process, each term (except for L_0) of the loss is actually the **KL divergence between 2 Gaussian distributions** which can be written explicitly as an L2-loss with respect to the means!

A direct consequence of the constructed forward process q, as shown by Sohl-Dickstein et al., is that we can sample $\mathbf{x}_t$ at any arbitrary noise level conditioned on $\mathbf{x}_0$ (since sums of Gaussians is also Gaussian). This is very convenient: we don't need to apply q repeatedly in order to sample $\mathbf{x}_t$. We have that

$$q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I})$$

with $\alpha_t := 1 - \beta_t$ and $\bar{\alpha}_t := \Pi_{s=1}^{t}\alpha_s$. Let's refer to this equation as the "nice property". This means we can sample Gaussian noise and scale it appropriately and add it to $\mathbf{x}_0$ to get $\mathbf{x}_t$ directly. Note that the $\bar{\alpha}_t$ are functions of the known β_t variance schedule and thus are also known and can be precomputed. This then allows us, during training, to **optimize random terms of the loss function L** (or in other words, to randomly sample t during training and optimize L_t).

Another beauty of this property, as shown in Ho et al. is that one can (after some math, for which we refer the reader to this excellent blog post) instead **reparametrize the mean to make the neural network learn (predict) the added noise (via a network $\epsilon_\theta(\mathbf{x}_t, t)$) for noise level t** in the KL terms which constitute the losses. This means that our neural network becomes a noise predictor, rather than a (direct) mean predictor. The mean can be computed as follows:

$$\mu_\theta(\mathbf{x}_t, t) = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(\mathbf{x}_t, t)\right)$$

The final objective function L_t then looks as follows (for a random time step t given $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$):

$$\|\epsilon - \epsilon_\theta(\mathbf{x}_t, t)\|^2 = \|\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{(1-\bar{\alpha}_t)}\epsilon, t)\|^2.$$

Here, $\mathbf{x}_0$ is the initial (real, uncorrupted) image, and we see the direct noise level t sample given by the fixed forward process. ϵ is the pure noise sampled at time step t, and $\epsilon_\theta(\mathbf{x}_t, t)$ is our neural network. The neural network is optimized using a simple mean squared error (MSE) between the true and the predicted Gaussian noise.

The Neural Network

The neural network needs to take in a noised image at a particular time step and return the predicted noise. Note that the predicted noise is a tensor that has the same size/resolution as the input image. So technically, the network takes in and outputs tensors of the same shape. What type of neural network can we use for this?

What is typically used here is very similar to that of an Autoencoder, which you may remember from typical "intro to deep learning" tutorials. Autoencoders have a so-called "bottleneck" layer in between the encoder and decoder. The encoder first encodes an image into a smaller hidden representation called the "bottleneck", and the decoder then decodes that hidden representation back into an actual image. This forces the network to only keep the most important information in the bottleneck layer.

In terms of architecture, the DDPM authors went for a **U-Net**, introduced by (Ronneberger et al., 2015) (which, at the time, achieved state-of-the-art results for medical image segmentation). This network, like any autoencoder, consists of a bottleneck in the middle that makes sure the network learns only the most important information. Importantly, it introduced residual connections between the encoder and decoder, greatly improving gradient flow (inspired by ResNet in He et al., 2015).

–

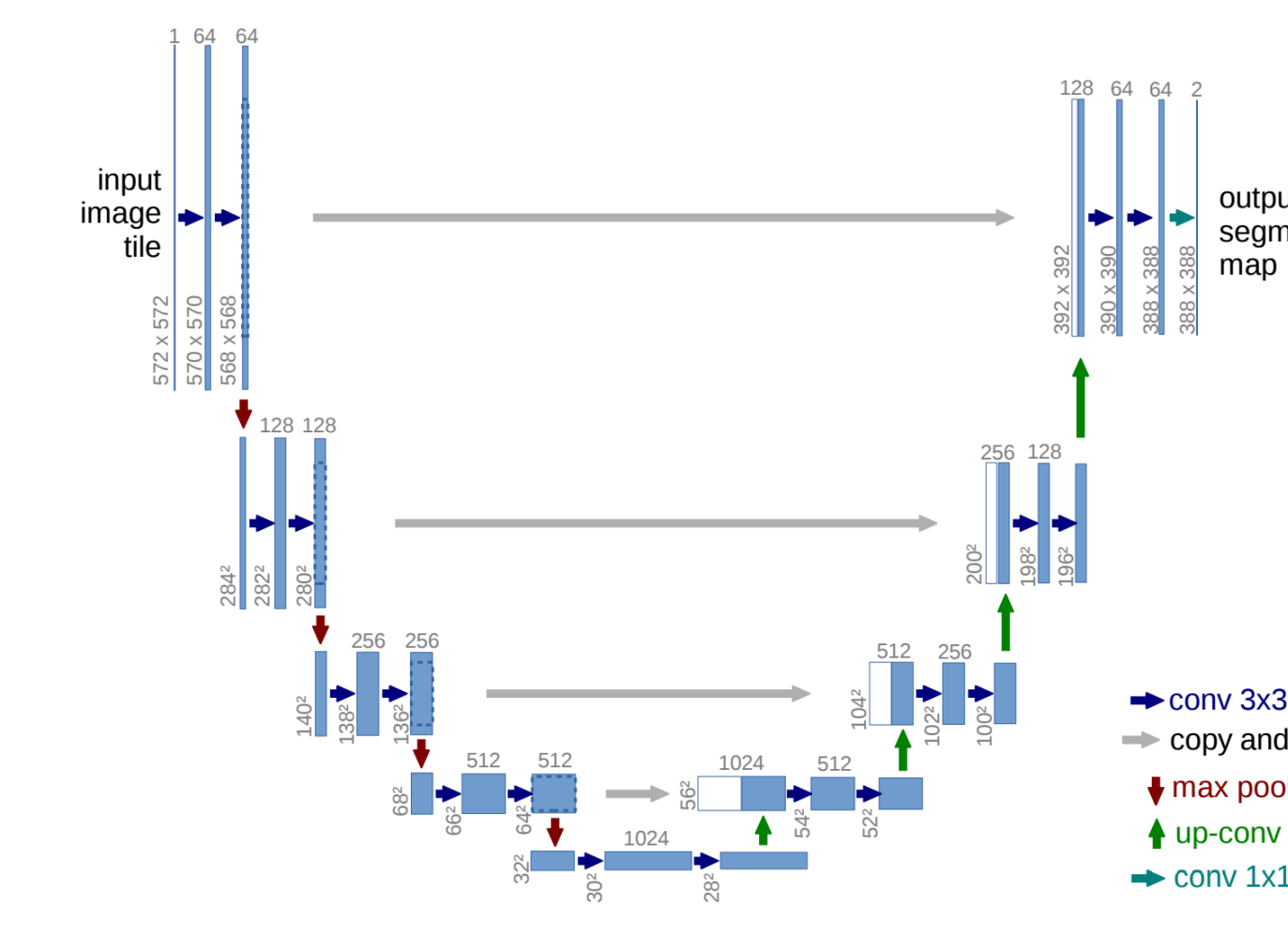

Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

Heading 3

Appendix

1. **ELBO:**

$$\begin{aligned}
-\log p_\theta(\mathbf{x}_0) &\leq -\log p_\theta(\mathbf{x}_0) + D_{\text{KL}}(q(\mathbf{x}_{1:T}|\mathbf{x}_0) \| p_\theta(\mathbf{x}_{1:T}|\mathbf{x}_0)) \\
&= -\log p_\theta(\mathbf{x}_0) + \mathbb{E}_{\mathbf{x}_{1:T}\sim q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\Big[\log \frac{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}{p_\theta(\mathbf{x}_{0:T})/p_\theta(\mathbf{x}_0)}\Big] \\
&= -\log p_\theta(\mathbf{x}_0) + \mathbb{E}_q\Big[\log \frac{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}{p_\theta(\mathbf{x}_{0:T})} + \log p_\theta(\mathbf{x}_0)\Big] \\
&= \mathbb{E}_q\Big[\log \frac{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}{p_\theta(\mathbf{x}_{0:T})}\Big] \\
\text{Let } L_{\text{VLB}} &= \mathbb{E}_{q(\mathbf{x}_{0:T})}\Big[\log \frac{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}{p_\theta(\mathbf{x}_{0:T})}\Big] \geq -\mathbb{E}_{q(\mathbf{x}_0)}\log p_\theta(\mathbf{x}_0)
\end{aligned}$$

$$\begin{aligned}
L_{\text{VLB}} &= \mathbb{E}_{q(\mathbf{x}_{0:T})}\Big[\log \frac{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}{p_\theta(\mathbf{x}_{0:T})}\Big] \\
&= \mathbb{E}_q\Big[\log \frac{\prod_{t=1}^T q(\mathbf{x}_t|\mathbf{x}_{t-1})}{p_\theta(\mathbf{x}_T)\prod_{t=1}^T p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}\Big] \\
&= \mathbb{E}_q\Big[-\log p_\theta(\mathbf{x}_T) + \sum_{t=1}^T \log \frac{q(\mathbf{x}_t|\mathbf{x}_{t-1})}{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}\Big] \\
&= \mathbb{E}_q\Big[-\log p_\theta(\mathbf{x}_T) + \sum_{t=2}^T \log \frac{q(\mathbf{x}_t|\mathbf{x}_{t-1})}{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)} + \log\frac{q(\mathbf{x}_1|\mathbf{x}_0)}{p_\theta(\mathbf{x}_0|\mathbf{x}_1)}\Big] \\
&= \mathbb{E}_q\Big[-\log p_\theta(\mathbf{x}_T) + \sum_{t=2}^T \log \Big(\frac{q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)}{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}\cdot\frac{q(\mathbf{x}_t|\mathbf{x}_0)}{q(\mathbf{x}_{t-1}|\mathbf{x}_0)}\Big) + \log\frac{q(\mathbf{x}_1|\mathbf{x}_0)}{p_\theta(\mathbf{x}_0|\mathbf{x}_1)}\Big] \\
&= \mathbb{E}_q\Big[-\log p_\theta(\mathbf{x}_T) + \sum_{t=2}^T \log \frac{q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)}{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)} + \sum_{t=2}^T \log\frac{q(\mathbf{x}_t|\mathbf{x}_0)}{q(\mathbf{x}_{t-1}|\mathbf{x}_0)} + \log\frac{q(\mathbf{x}_1|\mathbf{x}_0)}{p_\theta(\mathbf{x}_0|\mathbf{x}_1)}\Big] \\
&= \mathbb{E}_q\Big[-\log p_\theta(\mathbf{x}_T) + \sum_{t=2}^T \log \frac{q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)}{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)} + \log\frac{q(\mathbf{x}_T|\mathbf{x}_0)}{q(\mathbf{x}_1|\mathbf{x}_0)} + \log\frac{q(\mathbf{x}_1|\mathbf{x}_0)}{p_\theta(\mathbf{x}_0|\mathbf{x}_1)}\Big] \\
&= \mathbb{E}_q\Big[\log\frac{q(\mathbf{x}_T|\mathbf{x}_0)}{p_\theta(\mathbf{x}_T)} + \sum_{t=2}^T \log \frac{q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)}{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)} - \log p_\theta(\mathbf{x}_0|\mathbf{x}_1)\Big] \\
&= \mathbb{E}_q[\underbrace{D_{\text{KL}}(q(\mathbf{x}_T|\mathbf{x}_0) \| p_\theta(\mathbf{x}_T))}_{L_T} + \sum_{t=2}^T \underbrace{D_{\text{KL}}(q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) \| p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t))}_{L_{t-1}} \underbrace{- \log p_\theta(\mathbf{x}_0|\mathbf{x}_1)}_{L_0}]
\end{aligned}$$

Let's label each component in the variational lower bound loss separately:

$$\begin{aligned}
L_{\text{VLB}} &= L_T + L_{T-1} + \dots + L_0 \\
\text{where } L_T &= D_{\text{KL}}(q(\mathbf{x}_T|\mathbf{x}_0) \parallel p_\theta(\mathbf{x}_T)) \\
L_t &= D_{\text{KL}}(q(\mathbf{x}_t|\mathbf{x}_{t+1}, \mathbf{x}_0) \parallel p_\theta(\mathbf{x}_t|\mathbf{x}_{t+1})) \text{ for } 1 \leq t \leq T-1 \\
L_0 &= -\log p_\theta(\mathbf{x}_0|\mathbf{x}_1)
\end{aligned}$$

KL divergence: D(P || Q) = ∑ p(i) * log(p(i) / q(i))

2. **Loss function**

It is noteworthy that the reverse conditional probability is tractable when conditioned on $\mathbf{x}_0$:

$$q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) = \mathcal{N}(\mathbf{x}_{t-1}; \tilde{\boldsymbol{\mu}}(\mathbf{x}_t, \mathbf{x}_0), \tilde{\beta}_t\mathbf{I})$$

Using Bayes' rule, we have:

$$\begin{aligned}
q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) &= q(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{x}_0)\frac{q(\mathbf{x}_{t-1}|\mathbf{x}_0)}{q(\mathbf{x}_t|\mathbf{x}_0)} \\
&\propto \exp\Big(-\frac{1}{2}\Big(\frac{(\mathbf{x}_t - \sqrt{\alpha_t}\mathbf{x}_{t-1})^2}{\beta_t} + \frac{(\mathbf{x}_{t-1} - \sqrt{\bar{\alpha}_{t-1}}\mathbf{x}_0)^2}{1-\bar{\alpha}_{t-1}} - \frac{(\mathbf{x}_t - \sqrt{\bar{\alpha}_t}\mathbf{x}_0)^2}{1-\bar{\alpha}_t}\Big)\Big) \\
&= \exp\Big(-\frac{1}{2}\Big(\frac{\mathbf{x}_t^2 - 2\sqrt{\alpha_t}\mathbf{x}_t\mathbf{x}_{t-1} + \alpha_t\mathbf{x}_{t-1}^2}{\beta_t} + \frac{\mathbf{x}_{t-1}^2 - 2\sqrt{\bar{\alpha}_{t-1}}\mathbf{x}_0\mathbf{x}_{t-1} + \bar{\alpha}_{t-1}\mathbf{x}_0^2}{1-\bar{\alpha}_{t-1}} - \frac{(\mathbf{x}_t - \sqrt{\bar{\alpha}_t}\mathbf{x}_0)^2}{1-\bar{\alpha}_t}\Big)\Big) \\
&= \exp\Big(-\frac{1}{2}\Big(\Big(\frac{\alpha_t}{\beta_t} + \frac{1}{1-\bar{\alpha}_{t-1}}\Big)\mathbf{x}_{t-1}^2 - \Big(\frac{2\sqrt{\alpha_t}}{\beta_t}\mathbf{x}_t + \frac{2\sqrt{\bar{\alpha}_{t-1}}}{1-\bar{\alpha}_{t-1}}\mathbf{x}_0\Big)\mathbf{x}_{t-1} + C(\mathbf{x}_t, \mathbf{x}_0)\Big)\Big)
\end{aligned}$$

where $C(\mathbf{x}_t, \mathbf{x}_0)$ is some function not involving $\mathbf{x}_{t-1}$ and details are omitted. Following the standard Gaussian density function, the mean and variance can be parameterized as follows (recall that $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{i=1}^T \alpha_i$):

$$\begin{aligned}
\tilde{\beta}_t &= 1/\Big(\frac{\alpha_t}{\beta_t} + \frac{1}{1-\bar{\alpha}_{t-1}}\Big) = 1/\Big(\frac{\alpha_t - \bar{\alpha}_t + \beta_t}{\beta_t(1-\bar{\alpha}_{t-1})}\Big) = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\cdot\beta_t \\
\tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t, \mathbf{x}_0) &= \Big(\frac{\sqrt{\alpha_t}}{\beta_t}\mathbf{x}_t + \frac{\sqrt{\bar{\alpha}_{t-1}}}{1-\bar{\alpha}_{t-1}}\mathbf{x}_0\Big)/\Big(\frac{\alpha_t}{\beta_t} + \frac{1}{1-\bar{\alpha}_{t-1}}\Big) \\
&= \Big(\frac{\sqrt{\alpha_t}}{\beta_t}\mathbf{x}_t + \frac{\sqrt{\bar{\alpha}_{t-1}}}{1-\bar{\alpha}_{t-1}}\mathbf{x}_0\Big)\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\cdot\beta_t \\
&= \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\mathbf{x}_t + \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}\mathbf{x}_0
\end{aligned}$$

Thanks to the nice property, we can represent $\mathbf{x}_0 = \frac{1}{\sqrt{\bar{\alpha}_t}}(\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}_t)$ and plug it into the above equation and obtain:

$$\begin{aligned}
\tilde{\boldsymbol{\mu}}_t &= \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\mathbf{x}_t + \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}\frac{1}{\sqrt{\bar{\alpha}_t}}(\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}_t) \\
&= \frac{1}{\sqrt{\alpha_t}}\Big(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_t\Big)
\end{aligned}$$

Parameterization of L_t for Training Loss

Recall that we need to learn a neural network to approximate the conditioned probability distributions in the reverse diffusion process, $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t))$. We would like to train $\boldsymbol{\mu}_\theta$ to predict $\tilde{\boldsymbol{\mu}}_t = \frac{1}{\sqrt{\alpha_t}}\Big(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_t\Big)$. Because $\mathbf{x}_t$ is available as input at training time, we can reparameterize the Gaussian noise term instead to make it predict $\boldsymbol{\epsilon}_t$ from the input $\mathbf{x}_t$ at time step t:

$$\begin{aligned}
\boldsymbol{\mu}_\theta(\mathbf{x}_t, t) &= \frac{1}{\sqrt{\alpha_t}}\Big(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\Big) \\
\text{Thus } \mathbf{x}_{t-1} &= \mathcal{N}\Big(\mathbf{x}_{t-1}; \frac{1}{\sqrt{\alpha_t}}\Big(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\Big), \boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t)\Big)
\end{aligned}$$

The loss term L_t is parameterized to minimize the difference from $\tilde{\boldsymbol{\mu}}$:

$$\begin{aligned}
L_t &= \mathbb{E}_{\mathbf{x}_0, \boldsymbol{\epsilon}}\Big[\frac{1}{2\|\boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t)\|_2^2}\|\tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t, \mathbf{x}_0) - \boldsymbol{\mu}_\theta(\mathbf{x}_t, t)\|^2\Big] \\
&= \mathbb{E}_{\mathbf{x}_0, \boldsymbol{\epsilon}}\Big[\frac{1}{2\|\boldsymbol{\Sigma}_\theta\|_2^2}\Big\|\frac{1}{\sqrt{\alpha_t}}\Big(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_t\Big) - \frac{1}{\sqrt{\alpha_t}}\Big(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\Big)\Big\|^2\Big] \\
&= \mathbb{E}_{\mathbf{x}_0, \boldsymbol{\epsilon}}\Big[\frac{(1-\alpha_t)^2}{2\alpha_t(1-\bar{\alpha}_t)\|\boldsymbol{\Sigma}_\theta\|_2^2}\|\boldsymbol{\epsilon}_t - \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\|^2\Big] \\
&= \mathbb{E}_{\mathbf{x}_0, \boldsymbol{\epsilon}}\Big[\frac{(1-\alpha_t)^2}{2\alpha_t(1-\bar{\alpha}_t)\|\boldsymbol{\Sigma}_\theta\|_2^2}\|\boldsymbol{\epsilon}_t - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}_t, t)\|^2\Big]
\end{aligned}$$

Simplification

Empirically, Ho et al. (2020) found that training the diffusion model works better with a simplified objective that ignores the weighting term:

$$\begin{aligned}
L_t^{\text{simple}} &= \mathbb{E}_{t\sim[1,T], \mathbf{x}_0, \boldsymbol{\epsilon}_t}\Big[\|\boldsymbol{\epsilon}_t - \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\|^2\Big] \\
&= \mathbb{E}_{t\sim[1,T], \mathbf{x}_0, \boldsymbol{\epsilon}_t}\Big[\|\boldsymbol{\epsilon}_t - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}_t, t)\|^2\Big]
\end{aligned}$$

The final simple objective is:

$$L_{\text{simple}} = L_t^{\text{simple}} + C$$

where C is a constant not depending on θ.

Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \text{Uniform}(\{1, \dots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$\nabla_\theta \|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t)\|^2$
6: **until** converged

Algorithm 2 Sampling

1: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: **for** $t = T, \dots, 1$ **do**
3: $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
4: $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\Big(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\Big) + \sigma_t\mathbf{z}$
5: **end for**
6: **return** $\mathbf{x}_0$

Fig. 4. The training and sampling algorithms in DDPM (Image source: Ho et al. 2020)

3. **KL divergance between two gaussian distribution:**

Kullback-Leibler (KL) divergence is a measure of how one probability distribution diverges from a second, reference probability distribution. For Gaussian distributions, the KL divergence has a closed-form solution.

Assuming you have two Gaussian distributions, P and Q, with:

- P is a Gaussian distribution with mean μ_P and variance σ_P^2.
- Q is another Gaussian distribution with mean μ_Q and variance σ_Q^2.

The KL divergence from Q to P (denoted as $D_{KL}(P \parallel Q)$) can be calculated using the formula:

$$D_{KL}(P \parallel Q) = \log\frac{\sigma_Q}{\sigma_P} + \frac{\sigma_P^2 + (\mu_P - \mu_Q)^2}{2\sigma_Q^2} - \frac{1}{2}$$

This formula measures how much information is lost when Q is used to approximate P. Let's compute the KL divergence for specific means and variances of these distributions.

Conditional U-Net

Now that we've defined all building blocks (position embeddings, ResNet blocks, attention and group normalization), it's time to define the entire neural network. Recall that the job of the network $\epsilon_\theta(\mathbf{x}_t, t)$ is to take in a batch of noisy images and their respective noise levels, and output the noise added to the input. More formally:

- the network takes a batch of noisy images of shape `(batch_size, num_channels, height, width)` and a batch of noise levels of shape `(batch_size, 1)` as input, and returns a tensor of shape `(batch_size, num_channels, height, width)`

The network is built up as follows:

- first, a convolutional layer is applied on the batch of noisy images, and position embeddings are computed for the noise levels
- next, a sequence of downsampling stages are applied. Each downsampling stage consists of 2 ResNet blocks + groupnorm + attention + residual connection + a downsample operation
- at the middle of the network, again ResNet blocks are applied, interleaved with attention
- next, a sequence of upsampling stages are applied. Each upsampling stage consists of 2 ResNet blocks + groupnorm + attention + residual connection + an upsample operation
- finally, a ResNet block followed by a convolutional layer is applied.