Notes on Diffusion Models

1. Sampling from a normal distribution
2. How to learn a distribution p(x)
3. Distribution of natural images
4. Vanilla Diffusion Model
    1. Forward Diffusion Process
    2. Denoising Process
    3. Loss Function
    4. Training and Inference
    5. KPIs
5. Improvements
    1. Song et al (DDIM)
    2. Nichol et al
    3. Dhariwal et al
    4. Nichol et al (GLIDE)
    5. Ramesh et al (DALL-E 2)
    6. Rombach et al (Stable Diffusion)
6. UNet Architecture in Depth
7. ControlNet

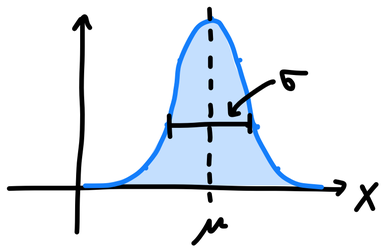
1 - Sampling from a normal distribution

Probability density function of a normal distribution:

$$\mathcal{N}(\mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)$$

MEAN $\in \mathbb{R}$

VARIANCE $\in \mathbb{R}^+$



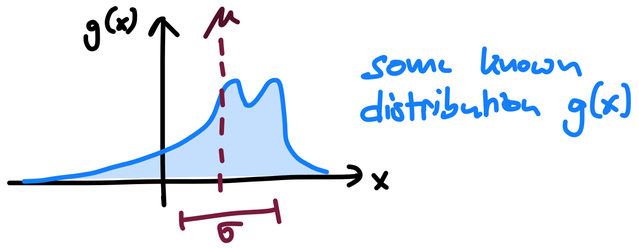
We are looking for a function f that randomly returns a number, such that the corresponding histogram after many executions of that function follows the pdf of the normal distribution.

Trick: Central Limit Theorem

> If $X_1, X_2, \ldots, X_n, \ldots$ are random samples drawn from a population with overall mean μ and finite variance σ^2, and if $\bar{X}_n$ is the sample mean of the first n samples, then the limiting form of the distribution, $Z = \lim_{n\to\infty}\left(\frac{\bar{X}_n - \mu}{\sigma_{\bar{X}}}\right)$, with $\sigma_{\bar{X}} = \sigma/\sqrt{n}$, is a standard normal distribution.[2]

https://en.wikipedia.org/wiki/Central_limit_theorem

In other words:



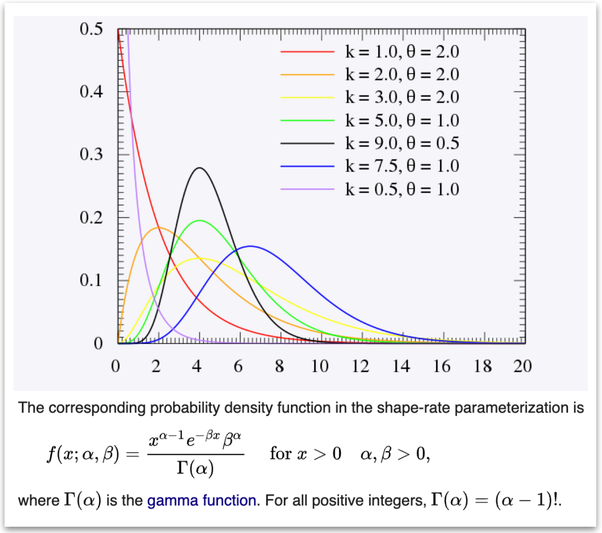
When sampling n times from g to get $\{x_i\}_{i=1\ldots n}$ we know that the average $m = \frac{1}{n}\sum_{i=1}^{n} x_i$ follows a normal distribution $\mathcal{N}(\mu, \frac{\sigma^2}{n})$.

If you want to draw from $\mathcal{N}(0,1)$, instead draw n times from g and calculate the following number:

$$r = \frac{\left(\frac{1}{n}\sum x_i\right) - \mu}{\sigma/\sqrt{n}}$$

2 - How to learn a distribution p(x)

Toy model: Gamma Distribution

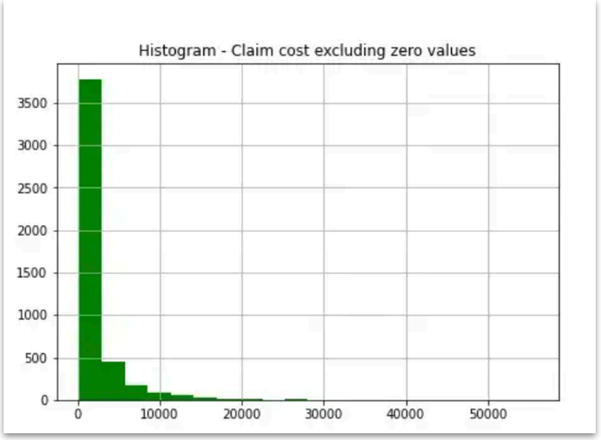


The corresponding probability density function in the shape-rate parameterization is

$$f(x;\alpha,\beta) = \frac{x^{\alpha-1}e^{-\beta x}\beta^{\alpha}}{\Gamma(\alpha)} \quad \text{for } x > 0 \quad \alpha,\beta > 0,$$

where $\Gamma(\alpha)$ is the gamma function. For all positive integers, $\Gamma(\alpha) = (\alpha-1)!$.

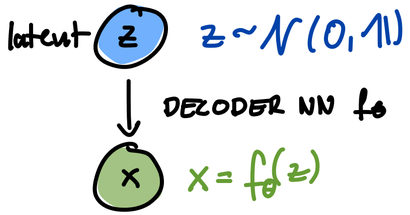
https://en.wikipedia.org/wiki/Gamma_distribution



https://medium.com/swlh/modeling-insurance-claim-severity-b449ac426c23

Let's learn this distribution, i.e find a function that randomly generates values such that the corresponding histogram after many generated values follows the probability density function of the gamma distribution.

- GRAPHICAL MODEL



- OPTIMIZATION VIA DATASET $\{x^{(i)}\}_{i=1..n}$

$$\theta^* = \arg\max_\theta \prod_{i=1}^{n} p_\theta(x^{(i)})$$

$$\Leftrightarrow \theta^* = \arg\max_\theta \sum_{i=1}^{n} \log p_\theta(x^{(i)})$$

- MODEL DEFINITION

$$p(z) = \mathcal{N}(0, 1I) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{z^2}{2}\right)$$

$$p_\theta(x|z) = \mathcal{N}(f_\theta(z), c\,1I) \qquad \mathbb{R}^+: \text{hyperparameter}$$

$$= \frac{1}{\sqrt{2\pi c}} \exp\left(-\frac{(x - f_\theta(z))}{2c}\right)$$

Derivation of the loss function:

$$\sum_{i=1}^{n} \log p_\theta(x^{(i)}) = \sum_{i=1}^{n} \log \int p_\theta(x^{(i)}|z)\, p(z)\, dz = \dots$$

- INTRACTABLE ⇒ Integration over whole latent space
- Trick: Knowing the mapping $x \to z$ would speed up

⇒ Introduce "encoder" to speed up training

$$x \to \mathcal{N}(g_\phi(x), \text{diag}(h_\phi(x))) = q_\phi(z|x)$$

$$\dots = \sum_{i=1}^{n} \log \int p_\theta(x^{(i)}|z)\, p(z)\, \frac{q_\phi(z|x)}{q_\phi(z|x)}\, dz$$

$$= \sum_{i=1}^{n} \log \int p_\theta(x^{(i)}|z)\, \frac{p(z)}{q_\phi(z|x)}\, q_\phi(z|x)\, dz$$

$$= \sum_{i=1}^{n} \log \left\langle p_\theta(x^{(i)}|z)\, \frac{p(z)}{q_\phi(z|x)} \right\rangle_{z \sim q_\phi(z|x)}$$

Jensen inequality

$$\geq \sum_{i=1}^{n} \left\langle \log \left(p_\theta(x^{(i)}|z)\, \frac{p(z)}{q_\phi(z|x)} \right) \right\rangle_{z \sim q_\phi(z|x)}$$

$$= \sum_{i=1}^{n} \left\langle \log p_\theta(x^{(i)}|z) \right\rangle_{z \sim q_\phi(z|x)} - \left\langle \log \frac{q_\phi(z|x)}{p(z)} \right\rangle_{z \sim q_\phi(z|x)}$$

$$= \sum_{i=1}^{n} \left\langle \log p_\theta(x^{(i)}|z) \right\rangle_{z \sim q_\phi(z|x)} - D_{KL}(q_\phi(z|x) \,\|\, p(z))$$

1st term — 2nd term

2 - How to learn a distribution p(x)

Explicit calculation of loss terms using Gaussian distributions:

$$\text{1st term} = \left\langle \log\left(\frac{1}{\sqrt{2\pi c}}\exp\left(-\frac{(x-f_\theta(z))^2}{2c}\right)\right)\right\rangle_{z\sim q_\phi(z|x)}$$

$$= \left\langle \underbrace{\log\frac{1}{\sqrt{2\pi c}}}_{:=\text{const.}} - \frac{[x-f_\theta(z)]^2}{2c}\right\rangle_{z\sim q_\phi(z|x)}$$

$$\text{2nd term} = D_{KL}\left(\mathcal{N}\left((\underbrace{\mu_1,\dots,\mu_d}_{g_\phi(x)})^T, \operatorname{diag}(\underbrace{\sigma_1^2,\dots,\sigma_d^2}_{h_\theta(x)})\right) \,\|\, \mathcal{N}(0,1)\right)$$

Wikipedia KL-Divergence

$$= \frac{1}{2}\sum_{j=1}^{d}\left(\sigma_j^2 + \mu_j^2 - 1 - \log\sigma_i^2\right)$$

Final loss:

$$\max \overset{!}{=} \sum_{i=1}^{n}\log p_\theta(x^{(i)}) \geq \sum_{i=1}^{n}\left\langle \underbrace{\log\frac{1}{\sqrt{2\pi c}}}_{:=\text{const.}} - \frac{[x-f_\theta(z)]^2}{2c}\right\rangle_{z\sim q_\phi(z|x)} - D_{KL}\left(q_\phi(z|x)\,\|\,p(z)\right)$$

$$\Leftrightarrow \quad \min \overset{!}{=} \sum_{i=1}^{n}\left(G\left\langle [x-f_\theta(z)]^2\right\rangle_{z\sim q(z|x)} + D_{KL}\left(q_\phi(z|x)\,\|\,p(z)\right)\right) = \mathcal{L}$$

Reconstruction Loss — Latent Loss

Assuming normal distributions results in the L2 norm, if we would model p(x|z) differently this term would be more complicated. See this blog post .

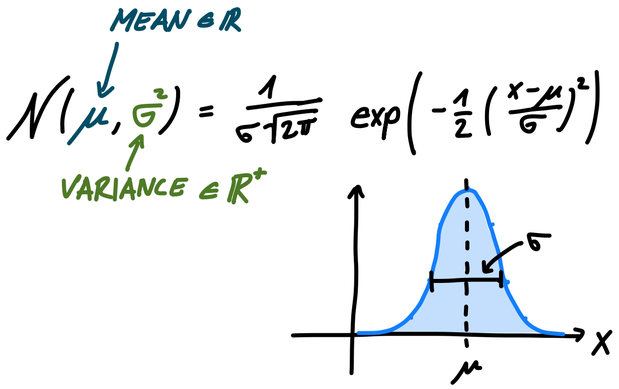
```python
# Model
# #####

class VAE(torch.nn.Module):
    def __init__(self, data_dim, latent_dim, n1=512, n2=256):
        super(VAE, self).__init__()

        # encoder part
        self.fc1 = torch.nn.Linear(data_dim, n1)
        self.fc2 = torch.nn.Linear(n1, n2)
        self.fc31 = torch.nn.Linear(n2, latent_dim)
        self.fc32 = torch.nn.Linear(n2, latent_dim)

        # decoder part
        self.fc4 = torch.nn.Linear(latent_dim, n2)
        self.fc5 = torch.nn.Linear(n2, n1)
        self.fc6 = torch.nn.Linear(n1, data_dim)

    def encoder(self, x):
        h = torch.relu(self.fc1(x))
        h = torch.relu(self.fc2(h))
        return self.fc31(h), self.fc32(h) # mu, log_var

    def sampling(self, mu, log_var):
        std = torch.exp(0.5*log_var)
        eps = torch.randn_like(std)
        return eps.mul(std).add_(mu) # return z sample

    def decoder(self, z):
        h = torch.relu(self.fc4(z))
        h = torch.relu(self.fc5(h))
        # return self.fc6(h)
        # return torch.sigmoid(self.fc6(h))
        return torch.relu(self.fc6(h))

    def forward(self, x):
        mu, log_var = self.encoder(x)
        z = self.sampling(mu, log_var)
        return self.decoder(z), mu, log_var

# Loss Function
# #############

def loss_function(X_tilde, X, mu, log_var, C=1):
    n = X.size(0)
    REC = (X_tilde - X).pow(2).sum()  / n
    KLD = 0.5 * torch.sum(mu.pow(2) + log_var.exp() - 1 - log_var) / n
    return C * REC + KLD, REC, KLD
```

…PyTorch implementation, turns out we developed a “Variational Autoencoder”.

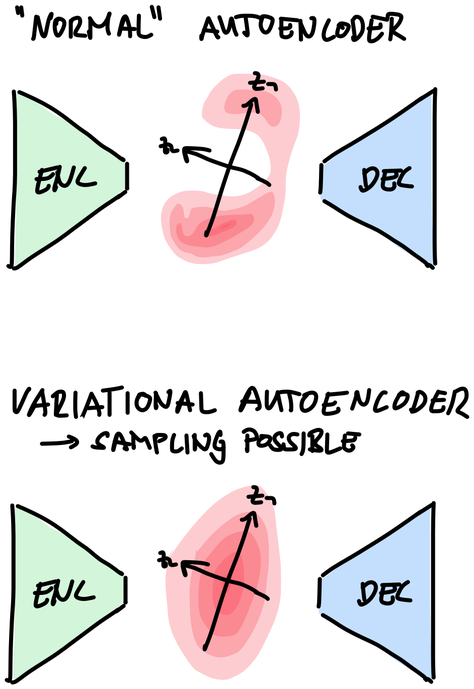
2 - How to learn a distribution p(x)

Some side notes:

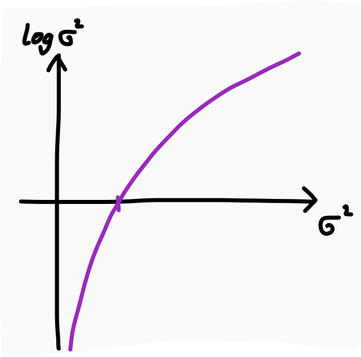
Gaussian notation: σ, σ², std & variance?



Autoencoder vs. Variational Autoencoder



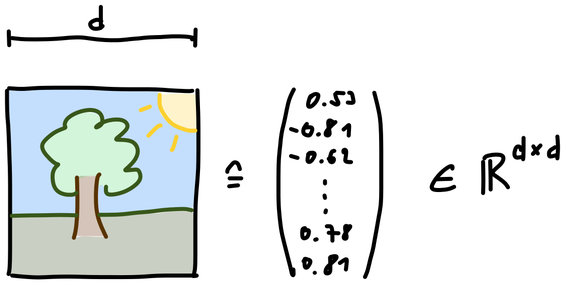
Why logvar?



$\log\sigma^2 \in \mathbb{R}$

$\sigma^2 = e^{\log\sigma^2} \in \mathbb{R}^+$

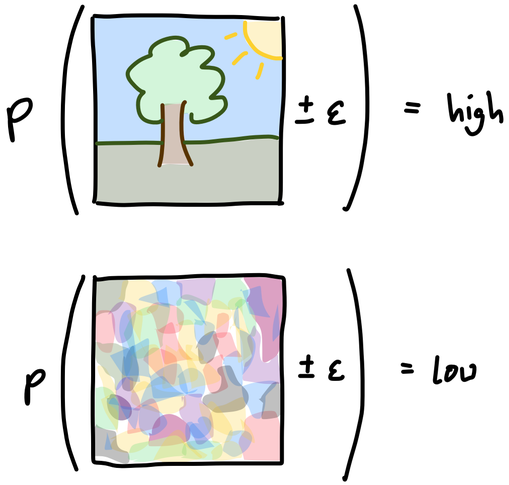
3 - Distribution of natural images



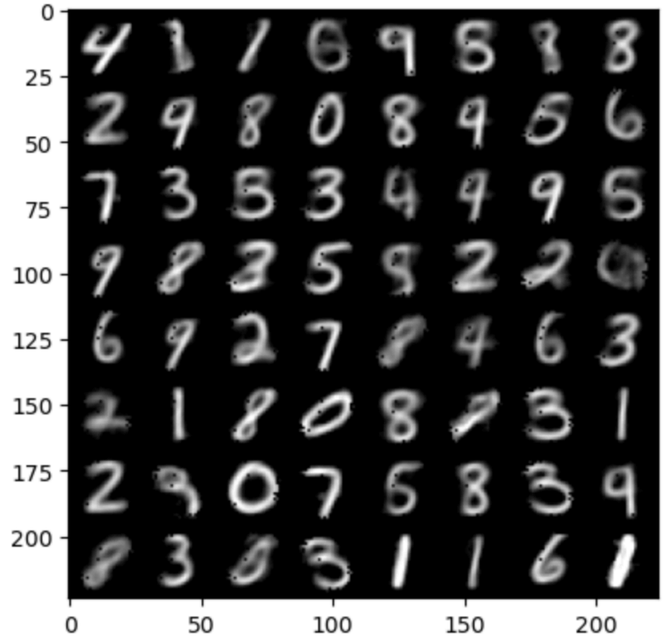
$\hat{=} \in \mathbb{R}^{d \times d}$

p: Distribution of natural images

$\Rightarrow$ very complicated




```python
n = 8
with torch.no_grad():
    z = torch.randn(n*n, LATENT_DIM).to(DEVICE)
    sample = vae.decoder(z).to(DEVICE)
    img = sample.view(n,n,28,28).cpu().numpy()
    img = np.vstack(np.dstack(img))
plt.imshow(img, cmap='gray', vmin=0, vmax=1)
plt.show()
```

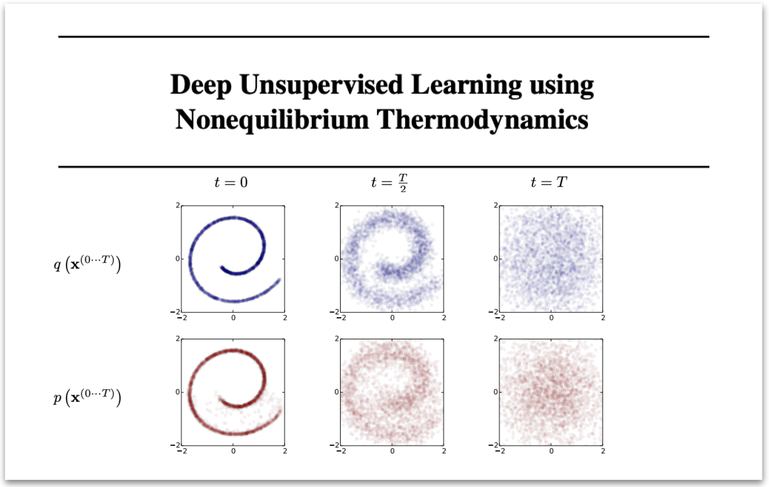


VAE from before trained on the MNIST dataset.

4 - Vanilla Diffusion Model

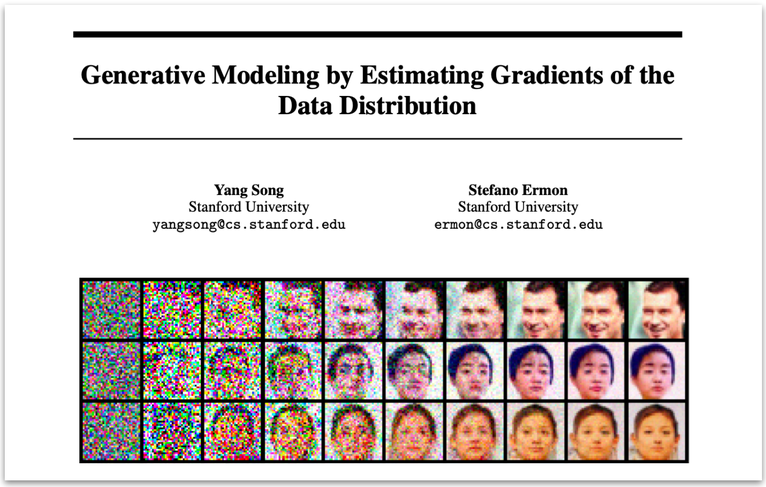
Pioneering papers:

Mar 2015: Sohl-Dickstein et al.



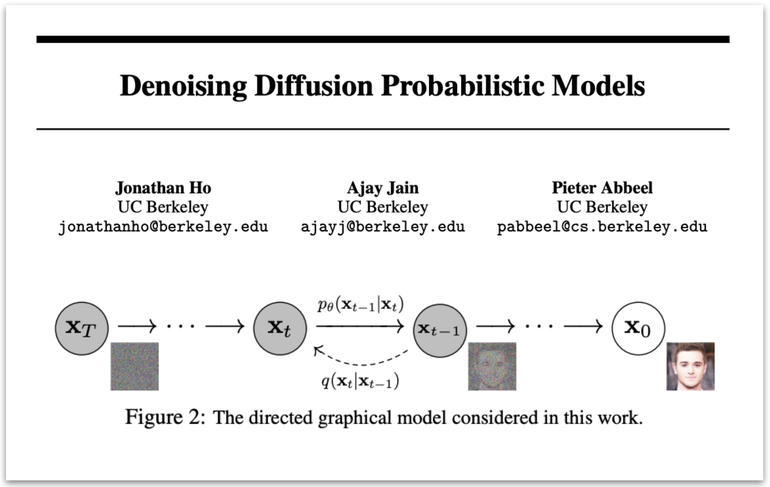
"The essential idea, inspired by non-equilibrium statistical physics, is to systematically and slowly destroy structure in a data distribution through an iterative forward diffusion process. We then learn a reverse diffusion process that restores structure in data, yielding a highly flexible and tractable generative model of the data."

Jul 2019: Song et al.

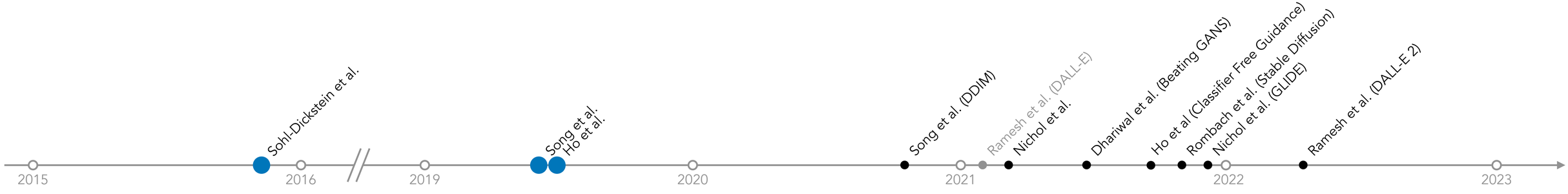


"We introduce a new generative model where samples are produced via Langevin dynamics using gradients of the data distribution estimated with score matching."

Jul 2019: Ho et al.



"This paper presents progress in diffusion probabilistic models [Sohl-Dickstein et al.]… When the diffusion consists of small amounts of Gaussian noise, it is sufficient to set the sampling chain transitions to conditional Gaussians too, allowing for a particularly simple neural network parameterisation."

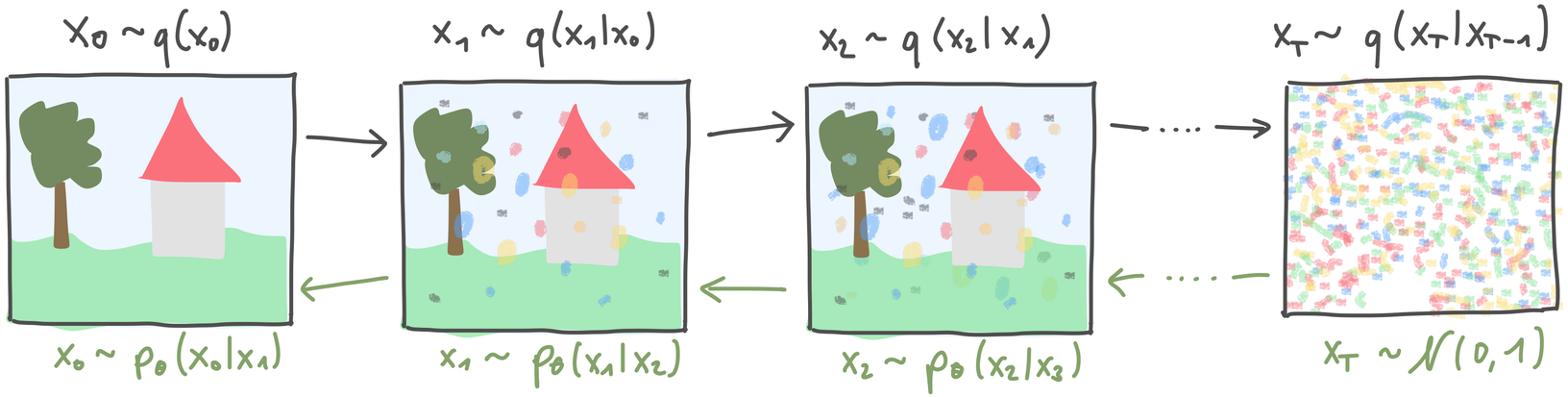


4 - Vanilla Diffusion Model

Basic idea:

DIFFUSION PROCESS

$$q(x_t \mid x_{t-1}) = \mathcal{N}(x_t; \mu, \sigma^2 \mathbb{1})$$



Real World Image

Isotropic Gaussian Noise

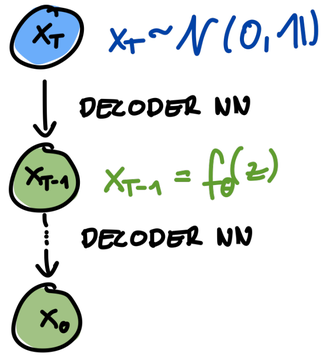
DENOISING PROCESS

$$q(x_{t-1} \mid x_t) = ? \approx p_\theta(x_{t-1} \mid x_t)$$

neural network parameters

no closer-form solution
→ no mean, no variance
→ no sampling

Graphical Model



4 - Vanilla Diffusion Model

Bayes does not work in practice…

denoising process

$$q(x_{t-1} \mid x_t) = \frac{q(x_t \mid x_{t-1}) \cdot q(x_{t-1})}{q(x_t)}$$

forward process

prior

we want to have a closed form solution of that distribution so that we can sample from

$$= \int q(x_t, x_{t-1})\, dx_{t-1} = \int q(x_t \mid x_{t-1})\, q(x_{t-1})\, dx_{t-1}$$

normalization

INTRACTABLE ⇕ HIGH DIMENSIONAL INTEGRATION

4.1 - Vanilla Diffusion Model - Forward Diffusion Process

Forward diffusion process:

Real data distribution: $x_0 \sim q(x)$

Definition of Forward Diffusion Process:

$$q(x_t | x_{t-1}) = \mathcal{N}(x_t; \sqrt{1-\beta_t}\, x_{t-1}, \beta_t \mathbb{1}) \quad \text{with } \beta_t \in (0,1)$$

($\sqrt{1-\beta_t}$: shifts μ towards 0 for large t)

$$q(x_{1:T} | x_0) = \prod_{t=1}^{T} q(x_t | x_{t-1})$$

($x_{1:T}$: $x_1, x_2, \ldots, x_T$)

after a sufficiently large number of steps T we arrive at isotropic Gaussian noise.

Nice property (closed form for direct sampling)

$$x_t = \sqrt{\alpha_t}\, x_{t-1} + \sqrt{1-\alpha_t}\, \varepsilon_{t-1}$$

$$= \sqrt{\alpha_t \alpha_{t-1}}\, x_{t-2} + \sqrt{1-\alpha_t \alpha_{t-1}}\, \bar{\varepsilon}_{t-2}$$

$$= \ldots$$

$$x_t = \sqrt{\bar{\alpha}_t}\, + \sqrt{1-\bar{\alpha}_t}\, \varepsilon$$

$$q(x_t | x_0) = \mathcal{N}(x_t; \sqrt{\bar{\alpha}_t}\, x_0, (1-\bar{\alpha}_t)\mathbb{1})$$

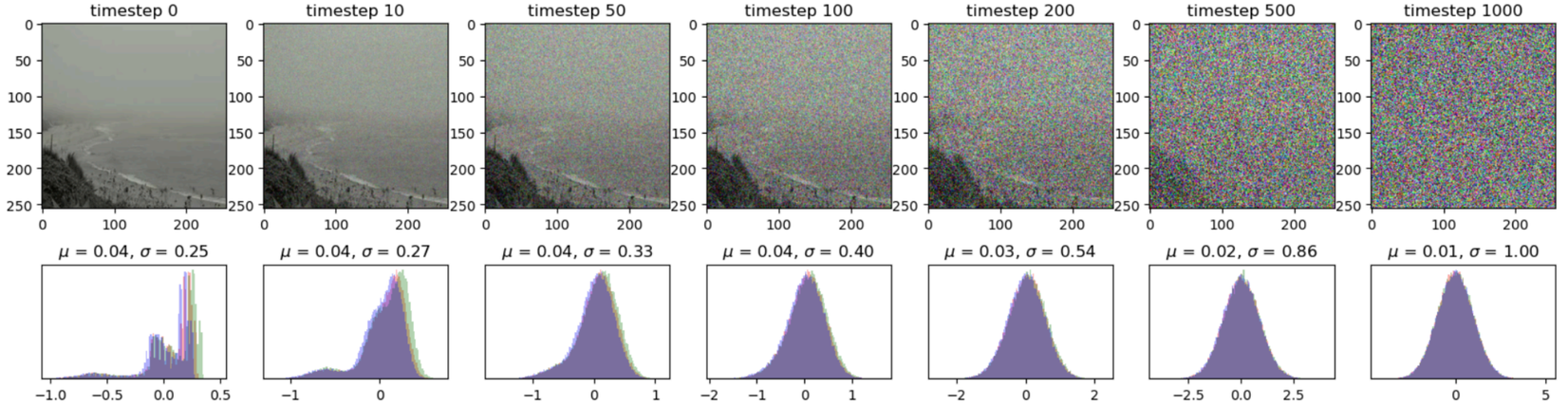
$\alpha_t = 1-\beta_t$, $\varepsilon_{t-1}, \varepsilon_{t-2}, \ldots \sim \mathcal{N}(0, \mathbb{1})$

use $\mathcal{N}(0, \sigma_1^2 \mathbb{1}) \oplus \mathcal{N}(0, \sigma_2^2 \mathbb{1}) = \mathcal{N}(0, (\sigma_1^2 + \sigma_2^2)\mathbb{1})$

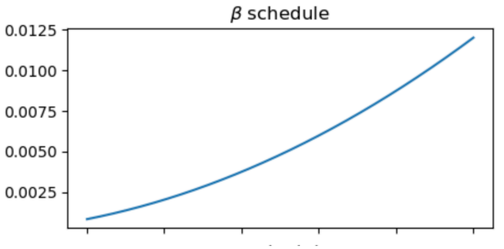
$$\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i \,, \quad \varepsilon \sim \mathcal{N}(0, \mathbb{1})$$

Example:

$T = 1000$

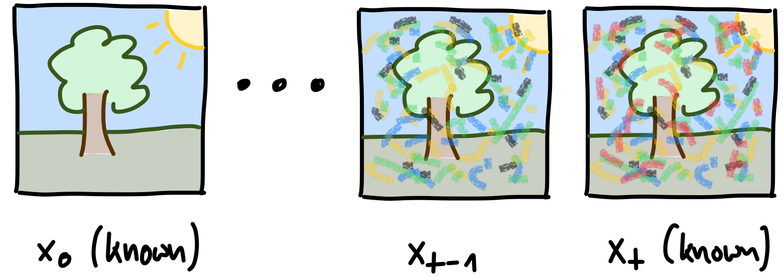


β_t increases quadratically from $\beta_1 = 0.85 \times 10^{-3}$ to $\beta_{1000} = 1.2 \times 10^{-3}$



4.1 - Vanilla Diffusion Model - Forward Diffusion Process

Important for later: $q(x_{t-1} \mid x_t, x_0)$ is tractable!



x_0 (known) $\quad x_{t-1} \quad x_t$ (known)

$$q(x_{t-1} \mid x_t, x_0) = \frac{q(x_t \mid x_{t-1}, x_0)}{q(x_t \mid x_{t-1})} \cdot \frac{q(x_{t-1} \mid x_0)}{q(x_t \mid x_0)} \qquad \text{Bayes rule}$$

$$\propto \exp\left[-\frac{1}{2} \cdot \frac{(x_t - \sqrt{\alpha_t}\, x_{t-1})^2}{\beta_t}\right] \cdot \exp\left[-\frac{1}{2} \frac{(x_{t-1} - \sqrt{\bar{\alpha}_{t-1}}\, x_0)^2}{1-\bar{\alpha}_{t-1}}\right] \cdot \exp\left[+\frac{1}{2} \frac{(x_t - \sqrt{\bar{\alpha}_t}\, x_0)^2}{1-\bar{\alpha}_t}\right]$$

$$= \exp\left[-\frac{1}{2}\left(\frac{x_t^2 - 2\sqrt{\alpha_t}\, x_t x_{t-1} + \alpha_t x_{t-1}^2}{\beta_t} + \frac{x_{t-1}^2 - 2\sqrt{\bar{\alpha}_{t-1}}\, x_{t-1} x_0 + \bar{\alpha}_{t-1} x_0^2}{1-\bar{\alpha}_{t-1}} - \frac{x_t^2 - 2\sqrt{\bar{\alpha}_t}\, x_0 x_t + \bar{\alpha}_t x_0^2}{1-\bar{\alpha}_t}\right)\right]$$

$$= \exp\left[-\frac{1}{2}\left(\underbrace{\left(\frac{\alpha_t}{\beta_t} + \frac{1}{1-\bar{\alpha}_{t-1}}\right)}_{1/\tilde{\beta}_t} x_{t-1}^2 - \underbrace{\left(\frac{2\sqrt{\alpha_t}}{\beta_t} x_t + \frac{2\sqrt{\bar{\alpha}_{t-1}}}{1-\bar{\alpha}_{t-1}} x_0\right)}_{2\tilde{\mu}_t/\tilde{\beta}_t} x_{t-1} + \underbrace{C(x_t, x_0)}_{\text{not important}}\right]\right]$$

$$\left\| \left(\frac{x-\tilde{\mu}}{\tilde{\beta}}\right)^2 = \frac{1}{\tilde{\beta}} x^2 - \frac{2\tilde{\mu}}{\tilde{\beta}} x + \frac{\tilde{\mu}^2}{\tilde{\beta}^2} \right.$$

$$= \exp\left[-\frac{1}{2}\left(\frac{x_{t-1} - \tilde{\mu}_t}{\tilde{\beta}_t}\right)\right]$$

$$\propto \mathcal{N}(x_{t-1}; \tilde{\mu}_t, \tilde{\beta}_t \mathbb{1})$$

4.1 - Vanilla Diffusion Model - Forward Diffusion Process

$$\tilde{\beta}_t = 1/\left(\frac{\alpha_t}{\beta_t} + \frac{1}{1-\bar{\alpha}_{t-1}}\right) = 1/\frac{\alpha_t(1-\bar{\alpha}_{t-1}) + \beta_t}{\beta_t(1-\bar{\alpha}_{t-1})} = \frac{(1-\bar{\alpha}_{t-1})\beta_t}{\alpha_t - \bar{\alpha}_t + \beta_t} = \frac{1-\bar{\alpha}_{t-1}}{1-\alpha_t}\beta_t$$

($1-\alpha_t$)

$$\tilde{\mu}_t = \left(\frac{\sqrt{\alpha_t}}{\beta_t}x_t + \frac{\sqrt{\bar{\alpha}_{t-1}}}{1-\bar{\alpha}_{t-1}}x_0\right)\cdot\tilde{\beta}_t = \left(\frac{\sqrt{\alpha_t}}{\beta_t}x_t + \frac{\sqrt{\bar{\alpha}_{t-1}}}{1-\bar{\alpha}_{t-1}}x_0\right)\cdot\frac{1-\bar{\alpha}_{t-1}}{1-\alpha_t}\cdot\beta_t$$

$$= \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\alpha_t}x_t + \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\alpha_t}x_0$$

($x_0 = \frac{1}{\sqrt{\bar{\alpha}_t}}(x_t - \sqrt{1-\bar{\alpha}_t}\,\varepsilon_t)$)

$$= \ldots = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon_t\right)$$

$$\Rightarrow \boxed{q(x_{t-1}\mid x_t, x_0) = \mathcal{N}\left(x_{t-1};\ \underbrace{\frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon_t\right)}_{\tilde{\mu}_t},\ \underbrace{\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t}_{\tilde{\beta}_t}\mathbb{1}\right)}$$

4.2 - Vanilla Diffusion Model - Denoising Process

Denoising process:

Small β_t

$\Rightarrow q(x_{t-1}|x_t)$ is Gaussian but still intractable

$\Rightarrow$ approximation $p_\theta(x_{t-1}, x_t)$ needed

Definition of denoising process:

$$p_\theta(x_{t-1}|x_t) := \mathcal{N}(x_{t-1}; \mu_\theta(x_t,t), \Sigma_\theta(x_t,t))$$

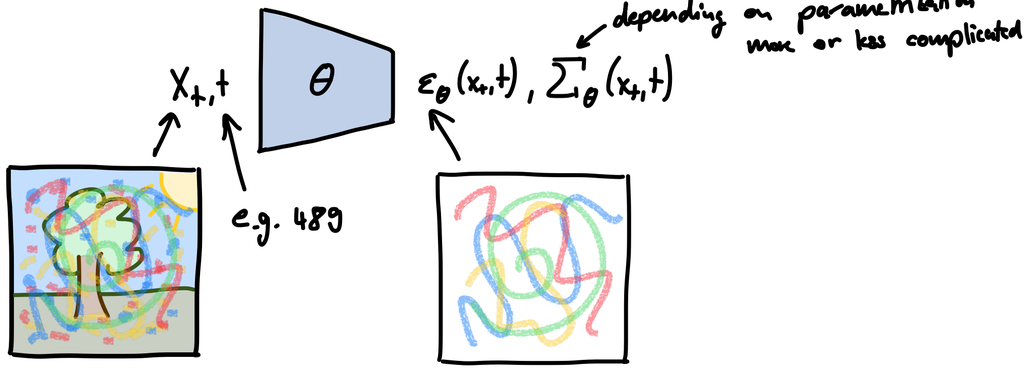
$$\rightarrow \quad p_\theta(x_{0:T}) = \underbrace{p_\theta(x_T)}_{= p(x_T) = \mathcal{N}(x_T; 0, \mathbb{1})} \prod_{t=1}^{T} p_\theta(x_{t-1}|x_t)$$

Instead of predicting μ_θ directly let's estimate the added noise $\varepsilon_\theta(x_t,t)$:

$$\mu_\theta(x_t,t) := \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon_\theta(x_t,t)\right)$$

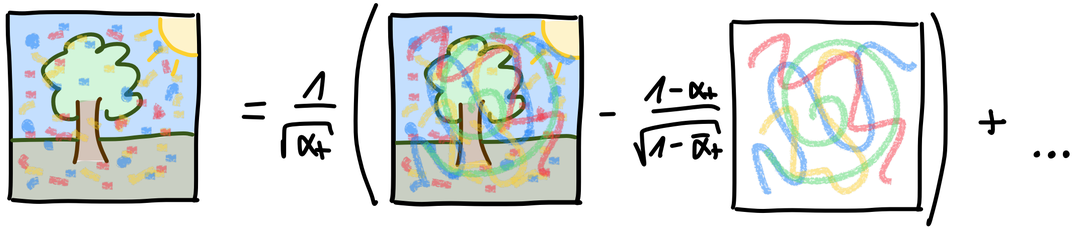
How one denoising step works in practice:

1. Estimate added mean noise ε_θ and covariance matrix Σ_θ:



2. Sample a denoised sample from the estimated distribution:

$$x_{t-1} \sim \mathcal{N}\left(x_{t-1}; \underbrace{\frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon_\theta(x_t,t)\right)}_{\mu_\theta(x_t,t)}, \Sigma_\theta(x_t,t)\right)$$



4.3 - Vanilla Diffusion Model - Loss Function

Loss function: Jansen inequality/ELBO/VLB

$$\max \overset{!}{=} \left\langle \log p_\theta(x_0) \right\rangle_{x_0 \sim q(x)}$$

$$= \left\langle \log \left(\int p_\theta(x_{0:T})\, dx_{1:T} \right) \right\rangle_{x_0 \sim q(x)} \quad \left| \; p_\theta(x_0) = \int p_\theta(x_0|x_1) \dots p(x_{T-1}|x_T)\, p(x_T)\, dx_1 \dots dx_T \right.$$

$$= \left\langle \log \left(\int q(x_{1:T}|x_0) \cdot \frac{p_\theta(x_{0:T})}{q(x_{1:T}|x_0)}\, dx_{1:T} \right) \right\rangle_{x_0 \sim q(x)} \quad \left| \; \text{Inserting a "1"} \right.$$

$$= \left\langle \log \left\langle \frac{p_\theta(x_{0:T})}{q(x_{1:T}|x_0)} \right\rangle_{x_{1:T} \sim q(x_{1:T}|x_0)} \right\rangle_{x_0 \sim q(x)}$$

$$\geq \left\langle \log \frac{p_\theta(x_{0:T})}{q(x_{1:T}|x_0)} \right\rangle_{x_{0:T} \sim q(x)} \quad \left| \; \text{Jansen inequality} \right.$$

$$\Rightarrow \boxed{\mathcal{L} = \left\langle \log \frac{q(x_{1:T}|x_0)}{p_\theta(x_{0:T})} \right\rangle_{x_{0:T} \sim q(x)}}$$

Also known as ELBO (Evidence Lower Bound) or VLB (Variational Lower Bound). Research “Variational Inference” for more details.

4.3 - Vanilla Diffusion Model - Loss Function

Loss function: Separation in different KL-divergence terms

$$\mathcal{L} = \left\langle \log \frac{q(x_{1:T}|x_0)}{p_\theta(x_{0:T})} \right\rangle_{x\sim q}$$

"every x_t in the equation is sampled from q"

$$= \left\langle \log \frac{\prod_{t=1}^{T} q(x_t|x_{t-1})}{p_\theta(x_T)\prod_{t=1}^{T} p_\theta(x_{t-1}|x_t)} \right\rangle_{x\sim q}$$

$$= \left\langle -\log p_\theta(x_T) + \sum_{t=1}^{T} \log \frac{q(x_t|x_{t-1})}{p_\theta(x_{t-1}|x_t)} \right\rangle_{x\sim q}$$

$$= \left\langle -\log p_\theta(x_T) + \left(\sum_{t=2}^{T} \log \frac{q(x_t|x_{t-1})}{p_\theta(x_{t-1}|x_t)}\right) + \log \frac{q(x_1|x_0)}{p_\theta(x_0|x_1)} \right\rangle_{x\sim q}$$

$$q(x_t|x_{t-1}) = q(x_t|x_{t-1}, x_0) \overset{\text{Bayes}}{=} \frac{q(x_{t-1}|x_t, x_0)\cdot q(x_t|x_0)}{q(x_{t-1}|x_0)}$$

$$= \left\langle -\log p_\theta(x_T) + \sum_{t=2}^{T} \log \left(\frac{q(x_{t-1}|x_t, x_0)}{p_\theta(x_{t-1}|x_t)} \cdot \frac{q(x_t|x_0)}{q(x_{t-1}|x_0)}\right) + \log \frac{q(x_1|x_0)}{p_\theta(x_0|x_1)} \right\rangle_{x\sim q}$$

$$= \left\langle -\log p_\theta(x_T) + \sum_{t=2}^{T} \log \frac{q(x_{t-1}|x_t, x_0)}{p_\theta(x_{t-1}|x_t)} + \sum_{t=2}^{T} \log \frac{q(x_t|x_0)}{q(x_{t-1}|x_0)} + \log \frac{q(x_1|x_0)}{p_\theta(x_0|x_1)} \right\rangle_{x\sim q}$$

$$\log\left(\frac{q(x_2|x_0)}{q(x_1|x_0)} \cdot \frac{q(x_3|x_0)}{q(x_2|x_0)} \cdots \frac{q(x_T|x_0)}{q(x_{T-1}|x_0)}\right) = \log \frac{q(x_T|x_0)}{q(x_1|x_0)}$$

$$= \left\langle -\log p_\theta(x_T) + \sum_{t=2}^{T} \log \frac{q(x_{t-1}|x_t, x_0)}{p_\theta(x_{t-1}|x_t)} + \log \frac{q(x_T|x_0)}{q(x_1|x_0)} + \log \frac{q(x_1|x_0)}{p_\theta(x_0|x_1)} \right\rangle_{x\sim q}$$

$$= \left\langle \log \frac{q(x_T|x_0)}{p_\theta(x_T)} + \sum_{t=2}^{T} \log \frac{q(x_{t-1}|x_t, x_0)}{p_\theta(x_{t-1}|x_t)} - \log p_\theta(x_0|x_1) \right\rangle_{x\sim q}$$

$$= \left\langle \log \frac{q(x_T|x_0)}{p_\theta(x_T)} \right\rangle_{x\sim q} + \sum_{t=2}^{T} \left\langle \log \frac{q(x_{t-1}|x_t, x_0)}{p_\theta(x_{t-1}|x_t)} \right\rangle_{x\sim q} - \left\langle \log p_\theta(x_0|x_1) \right\rangle_{x\sim q}$$

$$= \Big\langle \underbrace{D_{KL}\left(q(x_T|x_0) \,\|\, p_\theta(x_T)\right)}_{L_T} + \sum_{t=2}^{T} \underbrace{D_{KL}\left(q(x_{t-1}|x_t, x_0) \,\|\, p_\theta(x_{t-1}|x_t)\right)}_{L_{t-1}} \underbrace{- \log p_\theta(x_0|x_1)}_{L_0} \Big\rangle_{x\sim q}$$

4.3 - Vanilla Diffusion Model - Loss Function

Loss function: Explicit calculation of KL-divergence terms

- $\mathcal{L}_T = D_{KL}\left(q(x_T|x_0) \,\|\, p_\theta(x_T)\right)$

$q(x_T|x_0) = \mathcal{N}\left(x_T; \sqrt{\bar{\alpha}_T}\, x_0, (1-\bar{\alpha}_T)\mathbb{1}\right)$

$p_\theta(x_T) = p(x_T) = \mathcal{N}(x_T; 0, \mathbb{1})$

→ independent of θ → can be ignored in the loss

- $\mathcal{L}_0 = -\log p_\theta(x_0|x_1)$

$\sim \mathcal{N}(x_0; \mu_\theta(x_1, 1), \Sigma_\theta(x_1, 1))$

$= \frac{1}{2}(x_1 - \mu_\theta)^T \Sigma^{-1} (x_1 - \mu_\theta) - \log(\text{Prefactor})$

→ tractable

- $\mathcal{L}_{t-1} = D_{KL}\left(q(x_{t-1}|x_t, x_0) \,\|\, p_\theta(x_{t-1}|x_t)\right)$ for $2 \le t \le T$

$q(x_{t-1}|x_t, x_0) = \mathcal{N}\left(x_{t-1}; \underbrace{\frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon_t\right)}_{\tilde{\mu}_t(x_t, x_0)}, \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t \mathbb{1}\right)$

$p_\theta(x_{t-1}|x_t) = \mathcal{N}\left(x_{t-1}; \underbrace{\frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon_\theta(x_t, t)\right)}_{\mu_\theta(x_t, t)}, \Sigma_\theta(x_t, t)\right)$

KL-Divergence between two multivariant Gaussians has a closed form solution:

$$\mathcal{L}_{t-1} = \frac{1}{2\|\Sigma_\theta(x_t, t)\|^2} \left\| \tilde{\mu}_t(x_t, x_0) - \mu_\theta(x_t, t) \right\|^2 + \text{const.}$$

$$= \frac{1}{2\|\Sigma_\theta(x_t, t)\|^2} \cdot \left\| \frac{1}{\sqrt{\alpha_t}} \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \left(\varepsilon_t - \varepsilon_\theta(x_t, t)\right) \right\|^2 + \text{const.}$$

$$= \frac{(1-\alpha_t)^2}{2\alpha_t(1-\bar{\alpha}_t)\|\Sigma_\theta(x_t, t)\|^2} \left\| \varepsilon_t - \varepsilon_\theta(x_t, t) \right\|^2 + \text{const.}$$

→ tractable

4.3 - Vanilla Diffusion Model - Loss Function

Loss Function: Simplified Loss

3.4 Simplified training objective

With the reverse process and decoder defined above, the variational bound, consisting of terms derived from Eqs. (12) and (13), is clearly differentiable with respect to θ and is ready to be employed for training. However, we found it beneficial to sample quality (and simpler to implement) to train on the following variant of the variational bound:

$$L_{\text{simple}}(\theta) \coloneqq \mathbb{E}_{t,\mathbf{x}_0,\boldsymbol{\epsilon}}\left[\left\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t)\right\|^2\right] \tag{14}$$

where t is uniform between 1 and T. The $t = 1$ case corresponds to L_0 with the integral in the discrete decoder definition (13) approximated by the Gaussian probability density function times the bin width, ignoring σ_1^2 and edge effects. The $t > 1$ cases correspond to an unweighted version of Eq. (12), analogous to the loss weighting used by the NCSN denoising score matching model [55]. (L_T does not appear because the forward process variances β_t are fixed.) Algorithm 1 displays the complete training procedure with this simplified objective.

Since our simplified objective (14) discards the weighting in Eq. (12), it is a weighted variational bound that emphasizes different aspects of reconstruction compared to the standard variational bound [18, 22]. In particular, our diffusion process setup in Section 4 causes the simplified objective to down-weight loss terms corresponding to small t. These terms train the network to denoise data with very small amounts of noise, so it is beneficial to down-weight them so that the network can focus on more difficult denoising tasks at larger t terms. We will see in our experiments that this reweighting leads to better sample quality.

excerpt Ho et al.

independent of θ

$$L = \langle L_T + L_{T-1} + \dots + L_0 \rangle_{x_0 \sim q}$$

$$\Rightarrow L = \langle L_0 \rangle_{x_0 \sim q} + \sum_{t=2}^{T} \langle L_{t-1} \rangle_{x_t \sim q}$$

from before:

$$L_0 = \frac{1}{2}(x_0 - \mu_\theta)^T \Sigma^{-1}(x_0 - \mu_\theta) - \log(\text{Prefactor})$$

$$L_{t-1} = \underbrace{\frac{(1-\alpha_t)^2}{2\alpha_t(1-\bar{\alpha}_t)\|\Sigma_\theta(x_t,t)\|^2}}_{\text{weighting factor; will be dropped}} \| \varepsilon_t - \varepsilon_\theta(x_t,t) \|^2 + \text{const.}$$

$$\Rightarrow L^{\text{simple}} = \left\langle \| \varepsilon_t - \varepsilon_\theta(\underbrace{\sqrt{\bar{\alpha}_t}\, x_0 + \sqrt{1-\bar{\alpha}_t}\,\varepsilon}_{= x_t}, t) \|^2 \right\rangle_{x_0 \sim q,\, t \sim [0,\dots,T],\, \varepsilon \sim \mathcal{N}(0,1)}$$

4.4 - Vanilla Diffusion Model - Training and Inference

Algorithm 1 Training

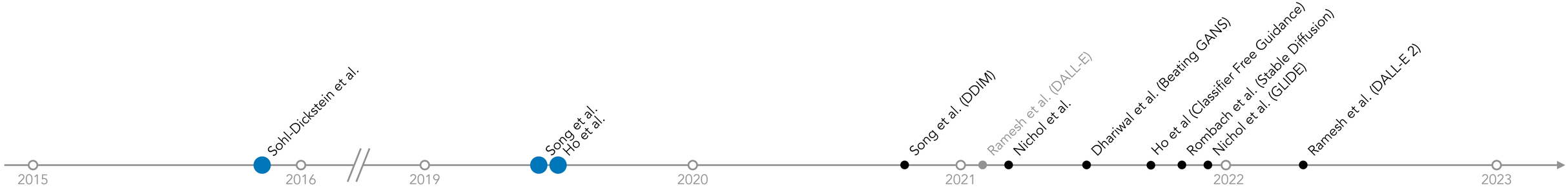
1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \text{Uniform}(\{1, \dots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0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

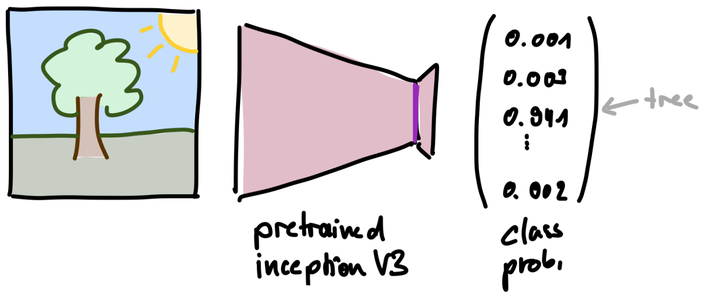
set to β_t or $\tilde{\beta}_t$

excerpt Ho et al.

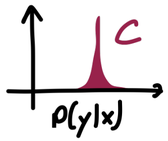


4.5 - Vanilla Diffusion Model - KPIs

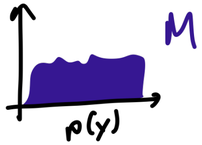
Inception Score (IS)



- Images with meaning full objects should have a "sharp" distribution $p(y|x)$.
  $\Rightarrow$ low entropy $p(y|x)$



- Model should output a variety of images.
  $\Rightarrow$ $p(y) = \int p(y|x)\,dx$ high entropy

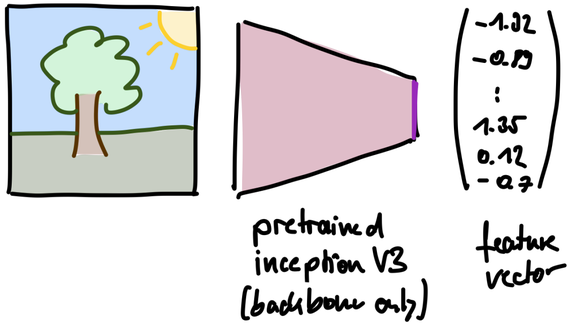


$\Rightarrow$ Combined metric: relative entropy

$$IS = \exp\left[\left\langle D_{KL}(c \,\|\, M) \right\rangle_{x \sim \text{all samples}} \right]$$

$IS \in [1, n_{classes}]$
bad ↑ ↖ good

Frechet Inception Distance (FID)



- Downside of IS: No comparison with real images
- Feature distribution of real images: μ, Σ
  Feature distribution of syn. images: $\tilde{\mu}, \tilde{\Sigma}$
  → Both distributions should be similar
  → Frechet distance should be small

$$FID = \|\mu - \tilde{\mu}\|^2 + Tr\left(\Sigma + \tilde{\Sigma} - 2\sqrt{\Sigma \cdot \tilde{\Sigma}}\right)$$

$FID \in [0, \infty]$
↑ good ↑ bad

Table 1: CIFAR10 results. NLL measured in bits/dim.

Model	IS	FID	NLL Test (Train)
Conditional			
EBM [11]	8.30	37.9	
JEM [17]	8.76	38.4	
BigGAN [3]	9.22	14.73	
StyleGAN2 + ADA (v1) [29]	**10.06**	**2.67**	
Unconditional			
Diffusion (original) [53]			≤ 5.40
Gated PixelCNN [59]	4.60	65.93	3.03 (2.90)
Sparse Transformer [7]			**2.80**
PixelIQN [43]	5.29	49.46	
EBM [11]	6.78	38.2	
NCSNv2 [56]		31.75	
NCSN [55]	8.87 ± 0.12	25.32	
SNGAN [39]	8.22 ± 0.05	21.7	
SNGAN-DDLS [4]	9.09 ± 0.10	15.42	
StyleGAN2 + ADA (v1) [29]	$\mathbf{9.74} \pm 0.05$	3.26	
Ours (L, fixed isotropic Σ)	7.67 ± 0.13	13.51	≤ 3.70 (3.69)
Ours (L_{simple})	9.46 ± 0.11	**3.17**	≤ 3.75 (3.72)

excerpt Ho et al.

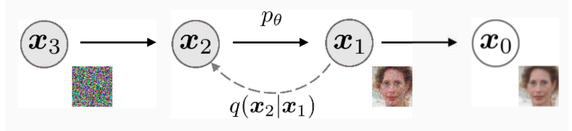
Links:
Blogpost: IS
Blogpost: FID

5.1 Improvements - Song et al (DDIM)

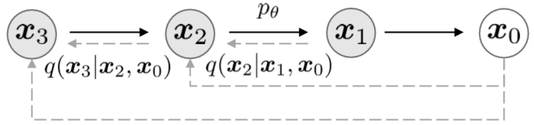
Denoising Diffusion Implicit Models

Jiaming Song, Chenlin Meng & Stefano Ermon
Stanford University
{tsong,chenlin,ermon}@cs.stanford.edu

Markovian forward process:



Non-Markovian forward process:



Definition of non-Markovian forward process:

$$q_\sigma(x_{1:T}|x_0) := q_\sigma(x_T|x_0)\prod_{t=2}^{T} q_\sigma(x_{t-1}|x_t,x_0)$$

$$q_\sigma(x_{t-1}|x_t,x_0) = \mathcal{N}\left(\sqrt{\alpha_{t-1}}x_0 + \sqrt{1-\alpha_{t-1}-\sigma_t^2}\,\frac{x_t-\sqrt{\alpha_t}x_0}{\sqrt{1-\alpha_t}},\ \sigma_t^2 I\right) \quad (*)$$

⇒ Bayes yields $q_\sigma(x_t|x_{t-1},x_0)$

⇒ One can prove $q_\sigma(x_t|x_0) = \mathcal{N}(\sqrt{\alpha_t}x_0, (1-\alpha_t)I)$

Resulting Loss function:

$$\mathbb{E}_{x_{0:T}\sim q_\sigma(x_{0:T})}\left[\log q_\sigma(x_T|x_0) + \sum_{t=2}^{T}\log q_\sigma(x_{t-1}|x_t,x_0) - \sum_{t=1}^{T}\log p_\theta^{(t)}(x_{t-1}|x_t) - \log p_\theta(x_T)\right]$$

- Best models for "normal" DDPM also minimise this loss
- Loss is independent of the exact time series of the forward procedure → allows us to skip steps

Sampling Process:

$x_t = \sqrt{\alpha_t}x_0 + \sqrt{1-\alpha_t}\,\varepsilon_t \rightarrow x_0 = (x_t - \sqrt{1-\alpha_t}\,\varepsilon)/\sqrt{\alpha_t}$ *Inserting in (*):*

$$x_{t-1} = \sqrt{\alpha_{t-1}}\underbrace{\left(\frac{x_t-\sqrt{1-\alpha_t}\epsilon_\theta^{(t)}(x_t)}{\sqrt{\alpha_t}}\right)}_{\text{"predicted } x_0\text{"}} + \underbrace{\sqrt{1-\alpha_{t-1}-\sigma_t^2}\cdot\epsilon_\theta^{(t)}(x_t)}_{\text{"direction pointing to } x_t\text{"}} + \underbrace{\sigma_t\epsilon_t}_{\text{random noise}}$$

Extreme cases:

$\sigma_t = 0 \Rightarrow$ Deterministic model (DDIM)

$\sigma_t = \sqrt{\frac{1-\alpha_{t-1}}{1-\alpha_t}}\sqrt{1-\frac{\alpha_t}{\alpha_{t-1}}}$

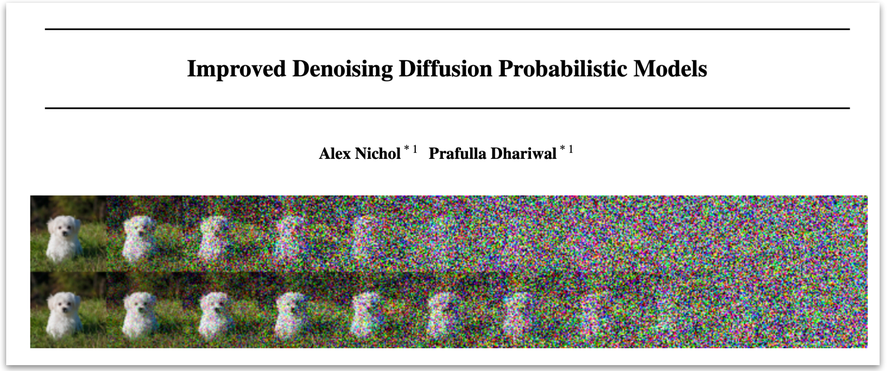
⇒ Markovian process of "normal" DDPMs

Experimental results:

	S	CIFAR10 (32 × 32) 10	20	50	100	1000	CelebA (64 × 64) 10	20	50	100	1000	
η	0.0	**13.36**	**6.84**	**4.67**	**4.16**	4.04	**17.33**	**13.73**	**9.17**	**6.53**	3.51	← DDIM
	0.2	14.04	7.11	4.77	4.25	4.09	17.66	14.11	9.51	6.79	3.64	
	0.5	16.66	8.35	5.25	4.46	4.29	19.86	16.06	11.01	8.09	4.28	
	1.0	41.07	18.36	8.01	5.78	4.73	33.12	26.03	18.48	13.93	5.98	} Ho et al
$\hat{\sigma}$		367.43	133.37	32.72	9.99	**3.17**	299.71	183.83	71.71	45.20	**3.26**	

FID scores

5.2 Improvements - Nichol et al



Improved Denoising Diffusion Probabilistic Models

Alex Nichol [*1] Prafulla Dhariwal [*1]

Main contributions:

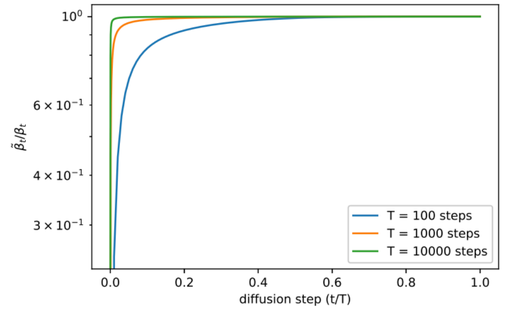
1. Learning Σ_θ and hybrid loss
2. Improved noise schedule β_t
3. Reduced gradient noise by improved sampling

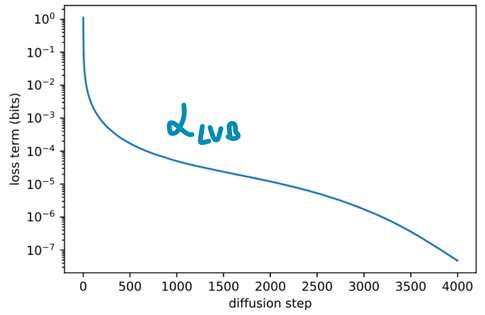
→ Nice side effect: Faster sampling speed

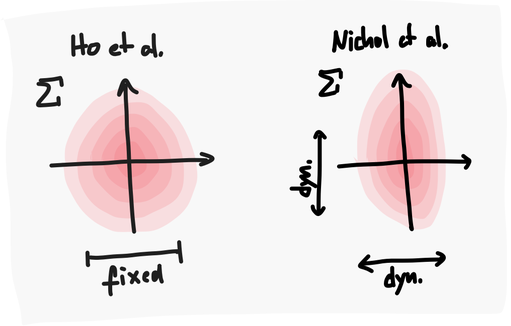
$$L_{\text{vlb}} := L_0 + L_1 + ... + L_{T-1} + L_T$$
$$L_0 := -\log p_\theta(x_0|x_1)$$
$$L_{t-1} := D_{KL}(q(x_{t-1}|x_t, x_0) \,||\, p_\theta(x_{t-1}|x_t))$$
$$L_T := D_{KL}(q(x_T|x_0) \,||\, p(x_T))$$

$$L_{\text{simple}} = E_{t,x_0,\epsilon}\left[||\epsilon - \epsilon_\theta(x_t, t)||^2\right]$$

Learning Σ_θ and hybrid loss







Ho et al: $\Sigma_\theta(x_t, t) = \sigma_t^2 \mathbb{1}$ ← hyperparameter, set to β_t or $\tilde{\beta}_t$

Recall: $q(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{1-\beta_t}\, x_{t-1}, \beta_t \mathbb{1})$ (diffusion)

$q(x_{t-1}|x_t, x_0) = \mathcal{N}(x_{t-1}; \tilde{\mu}(x_t, x_0), \tilde{\beta}_t \mathbb{1})$ (denoising)

$\tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$

$\Rightarrow \boxed{\tilde{\beta}_t \leq \sigma_t^2 \leq \beta_t}$

⇒ Real range is small, especially for large T

⇒ Hard to learn for a neural network (even at log-scale)

Ansatz:

$$\Sigma_\theta(x_t, t) = \exp\left(v \log \beta_t + (1-v)\log \tilde{\beta}_t\right) \qquad v \in \mathbb{R}^d$$

needs to be trained with:

$L_{\text{hybrid}} := L_{\text{simple}} + \lambda L_{\text{VLB}}$

L_{simple}: independent of Σ_θ

λ: $\frac{1}{1000}$

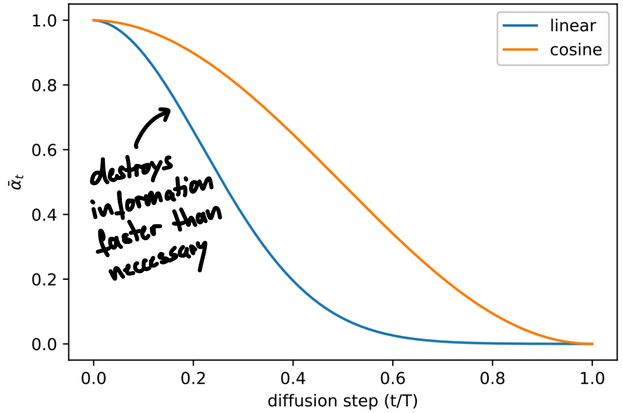
L_{VLB}: only important for small t; this is also where β_t and $\tilde{\beta}_t$ deviate the most!

5.2 Improvements - Nichol et al

Improved noise schedule β_t

linear

cosine

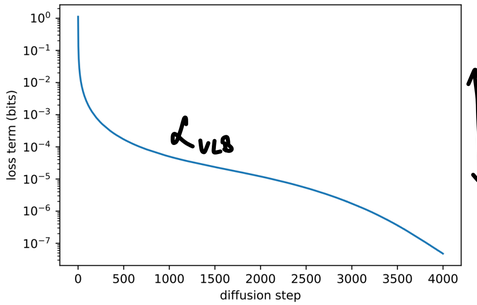


$$\bar{\alpha}_t = \frac{f(t)}{f(0)}$$

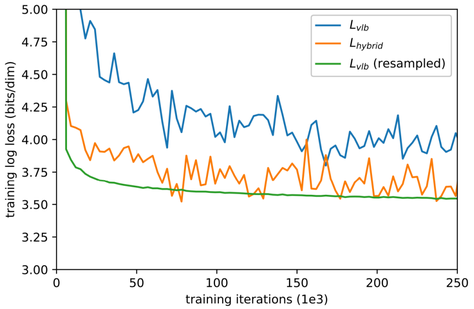
$$f(t) = \cos^2\left(\frac{t/T + s}{1+s} \cdot \frac{\pi}{2}\right)$$

"This choice was arbitrary"

Reduced gradient noise by improved sampling



spans multiple orders of magnitude ⇒ noisy loss landscape when sampling uniformly



$$L_{VLB} = \left\langle \frac{L_t}{p_t} \right\rangle_{t \sim p_t}$$

$$p_t \propto \sqrt{\langle L_t^2 \rangle} \quad \text{and} \quad \sum p_t = 1$$

example $L_{10} \gg L_{500}$

→ rescale both so that they have comparable values but sample $t = 20$ more often to compensate.

5.2 Improvements - Nichol et al

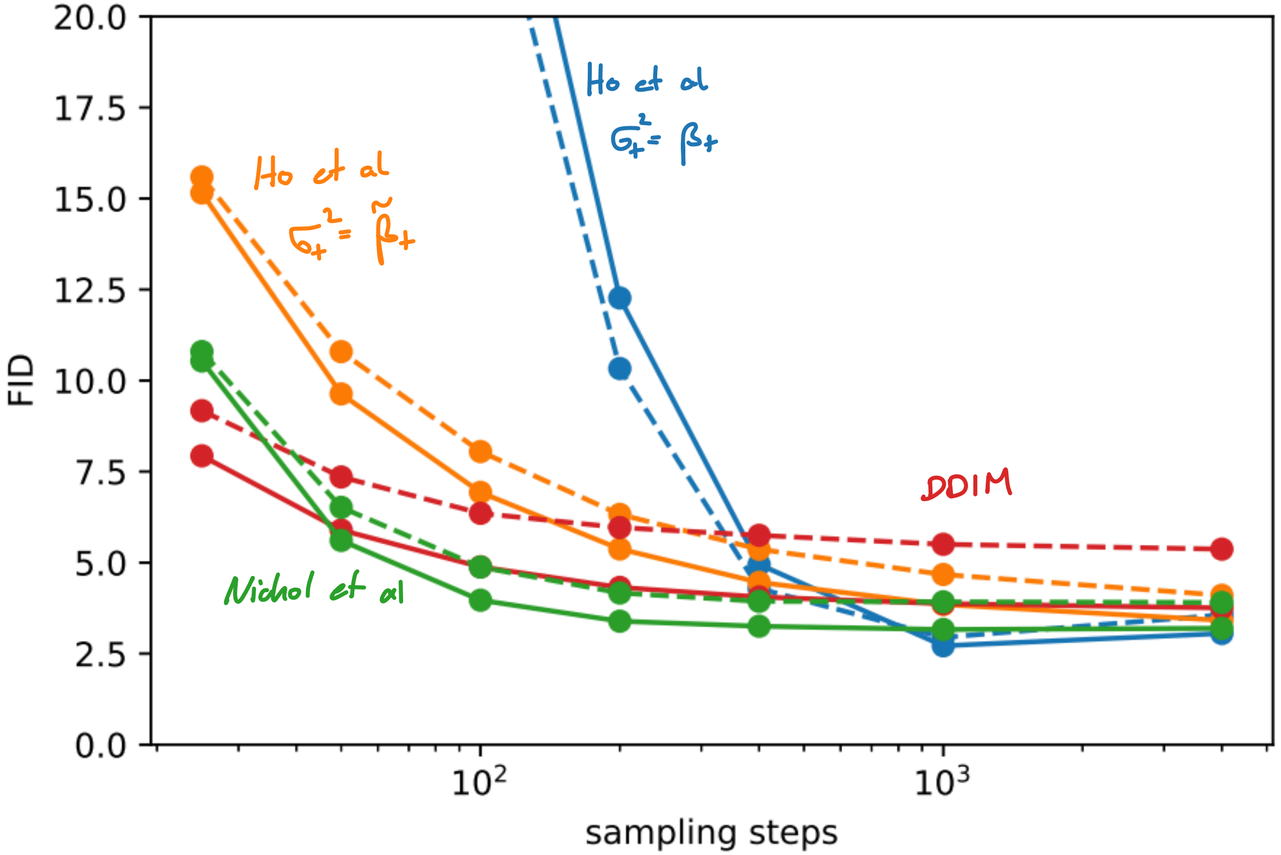
Nice side effect: Faster sampling speed

S = arbitrary subseries of $[1, 2, 3, \dots, T]$

$$\beta_{s_t} := 1 - \frac{\bar{\alpha}_{s_t}}{\bar{\alpha}_{s_{t-1}}}$$

$$\tilde{\beta}_{s_t} := \frac{1 - \bar{\alpha}_{s_{t-1}}}{1 - \bar{\alpha}_{s_t}} \beta_{s_t}$$

$$p(x_{s_{t-1}} \mid x_{s_t}) = \mathcal{N}(x_{s_{t-1}};\ \mu_\theta(x_{s_t}, s_t),\ \Sigma_\theta(x_{s_t}, s_t))$$



Idea:

$$\beta_t = 1 - \alpha_t$$

$$= 1 - \frac{\alpha_t \cdot \alpha_{t-1} \cdot \dots \cdot \alpha_0}{\alpha_{t-1} \cdot \alpha_{t-2} \cdot \dots \cdot \alpha_0}$$

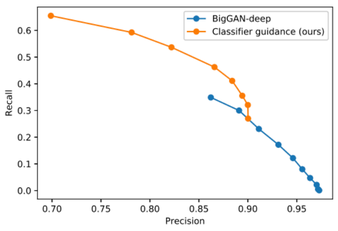
$$= 1 - \frac{\bar{\alpha}_t}{\bar{\alpha}_{t-1}}$$

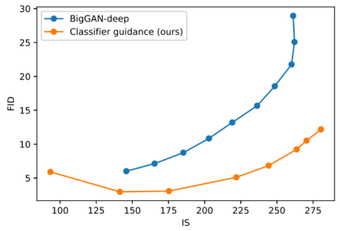
5.3 Improvements - Dhariwal et al

Diffusion Models Beat GANs on Image Synthesis

Prafulla Dhariwal*
OpenAI
prafulla@openai.com

Alex Nichol*
OpenAI
alex@openai.com





Model	FID	sFID	Prec	Rec
LSUN Bedrooms 256×256				
DCTransformer† [42]	6.40	6.66	0.44	**0.56**
DDPM [25]	4.89	9.07	0.60	0.45
IDDPM [43]	4.24	8.21	0.62	0.46
StyleGAN [27]	2.35	6.62	0.59	0.48
ADM (dropout)	**1.90**	**5.59**	**0.66**	0.51
LSUN Horses 256×256				
StyleGAN2 [28]	3.84	6.46	0.63	0.48
ADM	2.95	**5.94**	0.69	**0.55**
ADM (dropout)	**2.57**	6.81	**0.71**	**0.55**
LSUN Cats 256×256				
DDPM [25]	17.1	12.4	0.53	0.48
StyleGAN2 [28]	7.25	**6.33**	0.58	0.43
ADM (dropout)	**5.57**	6.69	**0.63**	**0.52**
ImageNet 64×64				
BigGAN-deep* [5]	4.06	3.96	**0.79**	0.48
IDDPM [43]	2.92	**3.79**	0.74	0.62
ADM	2.61	**3.77**	0.73	0.63
ADM (dropout)	**2.07**	4.29	0.74	**0.63**

Model	FID	sFID	Prec	Rec
ImageNet 128×128				
BigGAN-deep [5]	6.02	7.18	**0.86**	0.35
LOGAN† [68]	3.36			
ADM	5.91	**5.09**	0.70	**0.65**
ADM-G (25 steps)	5.98	7.04	0.78	0.51
ADM-G	**2.97**	**5.09**	0.78	0.59
ImageNet 256×256				
DCTransformer† [42]	36.51	8.24	0.36	**0.67**
VQ-VAE-2†‡ [51]	31.11	17.38	0.36	0.57
IDDPM‡ [43]	12.26	5.42	0.70	0.62
SR3†‡ [53]	11.30			
BigGAN-deep [5]	6.95	7.36	**0.87**	0.28
ADM	10.94	6.02	0.69	0.63
ADM-G (25 steps)	5.44	5.32	0.81	0.49
ADM-G	**4.59**	**5.25**	0.82	0.52
ImageNet 512×512				
BigGAN-deep [5]	8.43	8.13	**0.88**	0.29
ADM	23.24	10.19	0.73	**0.60**
ADM-G (25 steps)	8.41	9.67	0.83	0.47
ADM-G	**7.72**	**6.57**	0.87	0.42

Ablation studies on architecture design

- Started with UNet + global attention layer at 16x16 resolution + projection of timestamps embedded in each ResBlock.
- Ablation studies on depth vs with, number of attention heads, different resolutions of attention heads, using BigGAN Res blocks, etc…
- Conclusions:
  - More heads or few channels improve FID
  - 64 channels is best w.r.t wall clock time
  - Adaptive Gradient Normalisation Layer

Channels	Depth	Heads	Attention resolutions	BigGAN up/downsample	Rescale resblock	FID 700K	FID 1200K
160	2	1	16	✗	✗	15.33	13.21
128	4					-0.21	-0.48
		4				-0.54	-0.82
			32,16,8			-0.72	-0.66
				✓		-1.20	-1.21
					✓	0.16	0.25
160	2	4	32,16,8	✓	✗	**-3.14**	**-3.00**

Number of heads	Channels per head	FID
1		14.08
2		-0.50
4		-0.97
8		-1.17
	32	-1.36
	64	-1.03
	128	-1.08

Operation	FID
AdaGN	13.06
Addition + GroupNorm	15.08

In the rest of the paper, we use this final improved model architecture as our default:

- *variable width with 2 residual blocks per resolution,*
- *multiple heads with 64 channels per head,*
- *attention at 32, 16 and 8 resolutions,*
- *BigGAN residual blocks for up and downsampling,*
- *and adaptive group normalisation for injecting timestep and class embeddings into residual blocks.*

Main contributions:

1. Ablation studies on architecture design
2. Classifier guidance

5.3 Improvements - Dhariwal et al

Classifier Guidance

Motivation: GANs make heavy use of class labels. So how can we embed class labels into diffusion models?

$$p_{\theta,\phi}(x_t|x_{t+1}, y) = Z\, p_\theta(x_t|x_{t+1})\, p_\phi(y|x_t)$$

quite lengthy derivation

Z: normalization constant

p_ϕ: classifier

Recall:

$$p_\theta(x_t|x_{t+1}) = \mathcal{N}(\mu, \Sigma)$$

$$\Rightarrow \log p_\theta(x_t|x_{t+1}) = -\frac{1}{2}(x_t-\mu)^T\Sigma^{-1}(x_t-\mu) + C$$

After a lengthy calculation:

$$\log\left(p_\theta(x_t|x_{t+1})\, p_\phi(y|x_t)\right) = -\frac{1}{2}(x_t-\mu-\Sigma g)^T\Sigma^{-1}(x_t-\mu-\Sigma g) + C_2$$

$$g = \nabla_{x_t}\log(y|x_t)\big|_{x_t=\mu}$$

$$\Rightarrow \boxed{p_{\theta,\phi}(x_t|x_{t+1}, y) \sim \mathcal{N}(\mu+\Sigma g, \Sigma)}$$

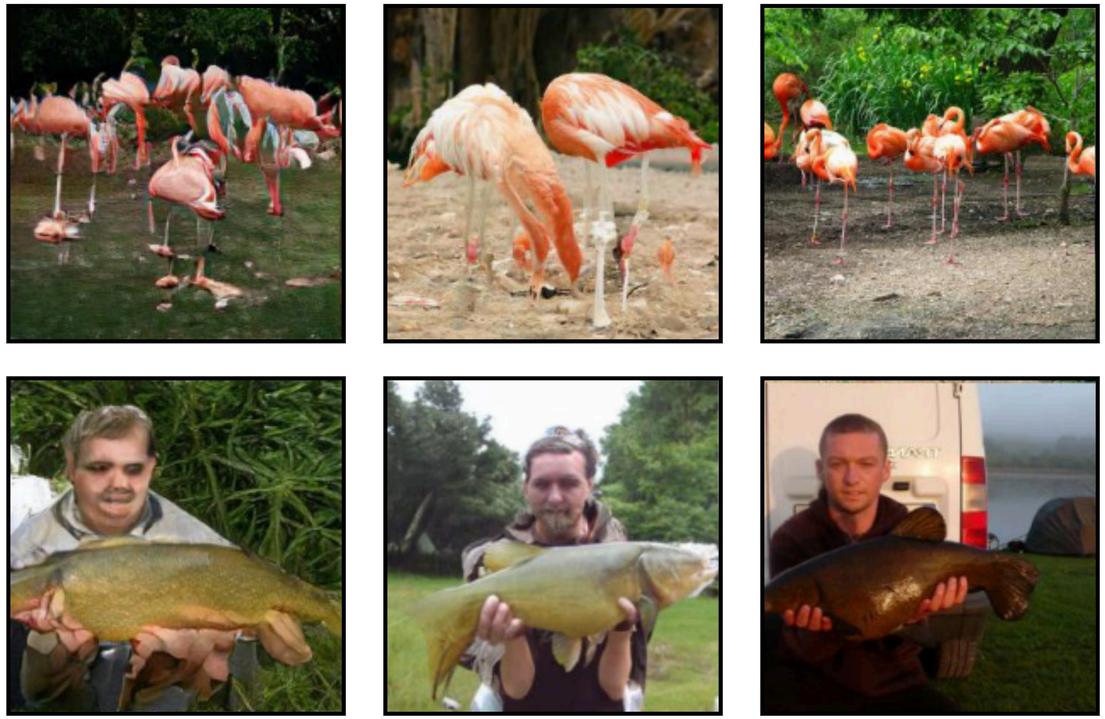
Algorithm 1 Classifier guided diffusion sampling, given a diffusion model $(\mu_\theta(x_t), \Sigma_\theta(x_t))$, classifier $p_\phi(y|x_t)$, and gradient scale s.

Input: class label y, gradient scale s
$x_T \leftarrow$ sample from $\mathcal{N}(0, \mathbf{I})$
for all t from T to 1 **do**
  $\mu, \Sigma \leftarrow \mu_\theta(x_t), \Sigma_\theta(x_t)$
  $x_{t-1} \leftarrow$ sample from $\mathcal{N}(\mu + s\Sigma\nabla_{x_t}\log p_\phi(y|x_t), \Sigma)$
end for
return x_0

Caution: Only works for stochastic models (e.g., DDPM). For DDIM there is an alternative solution. See the paper.

Conditional	Guidance	Scale	FID	sFID	IS	Precision	Recall
✗	✗		26.21	**6.35**	39.70	0.61	0.63
✗	✓	1.0	33.03	6.99	32.92	0.56	**0.65**
✗	✓	10.0	**12.00**	10.40	**95.41**	**0.76**	0.44
✓	✗		10.94	6.02	100.98	0.69	**0.63**
✓	✓	1.0	**4.59**	**5.25**	186.70	0.82	0.52
✓	✓	10.0	9.11	10.93	**283.92**	**0.88**	0.32

BigGAN | Diffusion Model | ImageNet Dataset



5.4 Improvements - Ho et al (Classifier Free Guidance)

Classifier-Free Diffusion Guidance

Jonathan Ho
Google Research
jonathanho@google.com

Tim Salimans
Google Research
salimans@google.com

Instead of training a separate classifier model, we choose to train an unconditional denoising diffusion model $p_\theta(\mathbf{z})$ parameterized through a score estimator $\epsilon_\theta(\mathbf{z}_\lambda)$ together with the conditional model $p_\theta(\mathbf{z}|\mathbf{c})$ parameterized through $\epsilon_\theta(\mathbf{z}_\lambda, \mathbf{c})$. We use a single neural network to parameterize both models, where for the unconditional model we can simply input zeros for the class identifier $\mathbf{c}$ when predicting the score, i.e. $\epsilon_\theta(\mathbf{z}_\lambda) = \epsilon_\theta(\mathbf{z}_\lambda, \mathbf{c} = \mathbf{0})$. We jointly train the unconditional and conditional models simply by randomly setting $\mathbf{c}$ to the unconditional class identifier.

Method	FID (↓)	IS (↑)
ADM [3]	2.07	-
CDM [6]	**1.48**	67.95
Ours, no guidance	1.80	53.71
Ours, with guidance		
$w = 0.1$	1.55	66.11
$w = 0.2$	2.04	78.91
$w = 0.3$	3.03	92.8
$w = 0.4$	4.30	106.2
$w = 0.5$	5.74	119.3
$w = 0.6$	7.19	131.1
$w = 0.7$	8.62	141.8
$w = 0.8$	10.08	151.6
$w = 0.9$	11.41	161
$w = 1.0$	12.6	170.1
$w = 2.0$	21.03	225.5
$w = 3.0$	24.83	250.4
$w = 4.0$	26.22	**260.2**

Figure 1: ImageNet 64x64 results

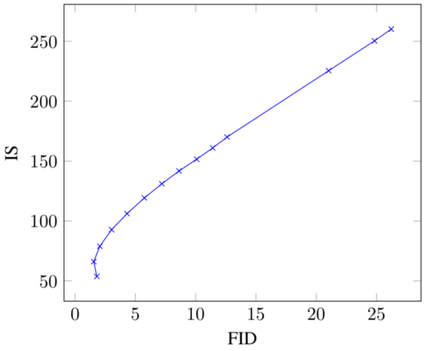
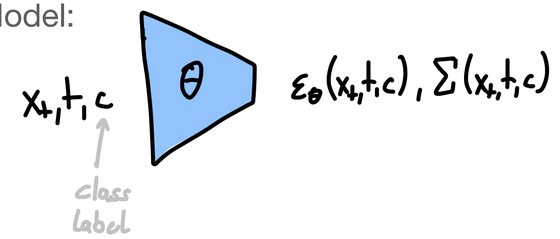
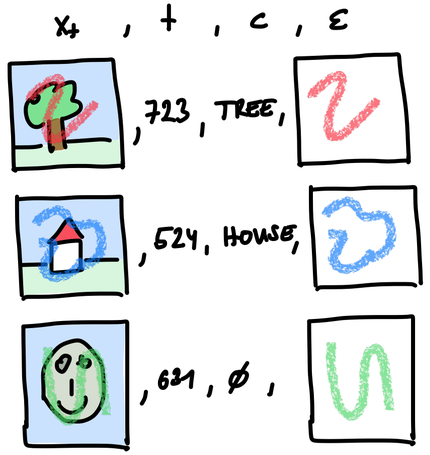


Figure 2: ImageNet 64x64 FID vs. IS

Model:



Training samples:

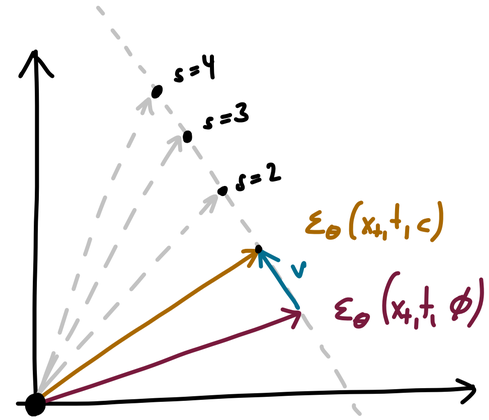


Estimated noise:

$$\tilde{\epsilon}_\theta(x_t,t,c) = (1+w)\epsilon_\theta(x_t,t,c) - w\epsilon_\theta(x_t,t,\emptyset)$$

$$w := s-1$$

$$= \epsilon_\theta(x_t,t,\emptyset) + s\underbrace{[\epsilon_\theta(x_t,t,c) - \epsilon_\theta(x_t,t,\emptyset)]}_{:=v}$$



5.4 Improvements - Nichol et al (GLIDE)

GLIDE: Towards Photorealistic Image Generation and Editing with Text-Guided Diffusion Models

Alex Nichol * Prafulla Dhariwal * Aditya Ramesh * Pranav Shyam Pamela Mishkin Bob McGrew
Ilya Sutskever Mark Chen

"a hedgehog using a calculator"

"a corgi wearing a red bowtie and a purple party hat"

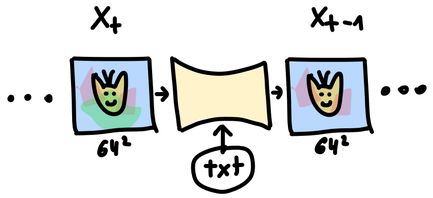
"robots meditating in a vipassana retreat"

Main contributions:

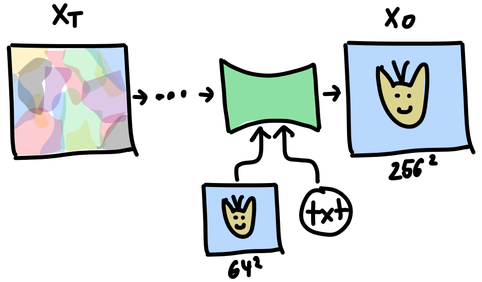
1. Text conditional guidance
2. Iterative generation by (Improved) impainting

Architecture:

① text-conditional diffusion model



② text-conditional upscaling diffusion model (actually from Nichol et al.)



③ text-encoder

We adopt the ADM model architecture proposed by Dhariwal & Nichol (2021), but augment it with text conditioning information. For each noised image x_t and corresponding text caption c, our model predicts $p(x_{t-1}|x_t, c)$. To condition on the text, we first encode it into a sequence of K tokens, and feed these tokens into a Transformer model (Vaswani et al., 2017). The output of this transformer is used in two ways: first, the final token embedding is used in place of a class embedding in the ADM model; second, the last layer of token embeddings (a sequence of K feature vectors) is separately projected to the dimensionality of each attention layer throughout the ADM model, and then concatenated to the attention context at each layer.

Training:

- Pre-training on DALL-E dataset.
- Fine tuning for classifier-free guidance by randomly dropping 20% of the image descriptions.
- Authors also tried CLIP guidance but classifier-free guidance yields better results.

Improved Denoising Diffusion Probabilistic Models

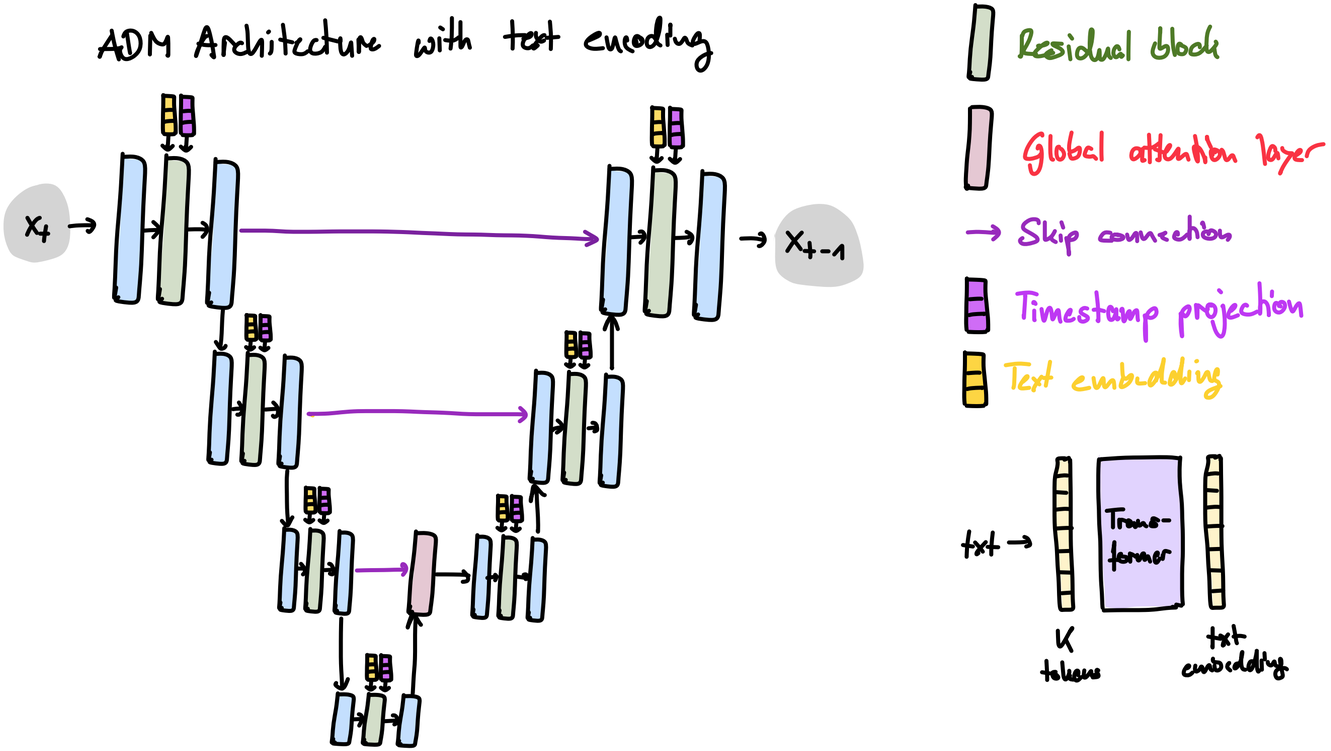
C. Sample Quality on ImageNet 256×256

We trained two models on class conditional ImageNet 256×256. The first is a usual diffusion model that directly models the 256×256 images. The second model reduces compute by chaining a pretrained 64×64 model $p(x_{64}|y)$ with another upsampling diffusion model $p(x_{256}|x_{64}, y)$ to upsample images to 256×256. For the upsampling model, the downsampled image x_{64} is passed as extra conditioning input to the UNet. This is similar to VQ-VAE-2 (Razavi et al., 2019), which uses two stages of priors at different latent resolutions to more efficiently learn global and local features. The linear schedule worked better for 256×256 images, so we used that for these results. Table 5 summarizes our results. For VQ-VAE-2, we use the FIDs reported in (Ravuri & Vinyals, 2019). Diffusion models still obtain the best FIDs for a likelihood-based model, and close the gap to GANs considerably.

MODEL	FID
VQ-VAE-2 ((Razavi et al., 2019), two-stage)	38.1
Improved Diffusion (ours, single-stage)	31.5
Improved Diffusion (ours, two-stage)	**12.3**
BigGAN (Brock et al., 2018)	7.7
BigGAN-deep (Brock et al., 2018)	**7.0**

5.4 Improvements - Nichol et al (GLIDE)

Rough schema of the architecture (however, not 100% sure):



How to find the architecture….

Nichol et al: *"We use the same model architecture as the ImageNet 64 × 64 model from Dhariwal & Nichol (2021), but scale the model width to 512 channels"*

↓

Dharwal et al: *"Ho et al. [25] introduced the UNet architecture for diffusion models"*

↓

Ho et al.: *"…we use a U-Net backbone similar to an unmasked PixelCNN++…"*

↓

Pixel CNN++: *"PixelCNN++: Improving the PixelCNN…"*

↓

"Oil painting of a frustrated person sitting in his office, throwing around scientific articles."

5.5 Improvements - Ramesh et al (DALL-E 2)

Hierarchical Text-Conditional Image Generation with CLIP Latents

Aditya Ramesh* OpenAI aramesh@openai.com

Prafulla Dhariwal* OpenAI prafulla@openai.com

Alex Nichol* OpenAI alex@openai.com

Casey Chu* OpenAI casey@openai.com

Mark Chen OpenAI mark@openai.com

Abstract

Contrastive models like CLIP have been shown to learn robust representations of images that capture both semantics and style. To leverage these representations for image generation, we propose a two-stage model: a prior that generates a CLIP image embedding given a text caption, and a decoder that generates an image conditioned on the image embedding. We show that explicitly generating image representations improves image diversity with minimal loss in photorealism and caption similarity. Our decoders conditioned on image representations can also produce variations of an image that preserve both its semantics and style, while varying the non-essential details absent from the image representation. Moreover, the joint embedding space of CLIP enables language-guided image manipulations in a zero-shot fashion. We use diffusion models for the decoder and experiment with both autoregressive and diffusion models for the prior, finding that the latter are computationally more efficient and produce higher-quality samples.

Main contributions:

1. Leveraging powerful CLIP embeddings

Main idea:

- *"CLIP embeddings are robust to image distribution shift, have impressive zero-shot capabilities, and have been fine-tuned to achieve state-of-the-art results on a wide variety of vision and language tasks."*
- *"Diffusion models have emerged as a promising generative modeling framework, pushing the state-of-the-art on image and video generation tasks"*

"CLIP + diffusion model = DALLE 2"

Two staged model:

- Prior $P(z_i|y)$
- Decoder $P(x|z_i,y)$

y: caption

z_i: image encoding

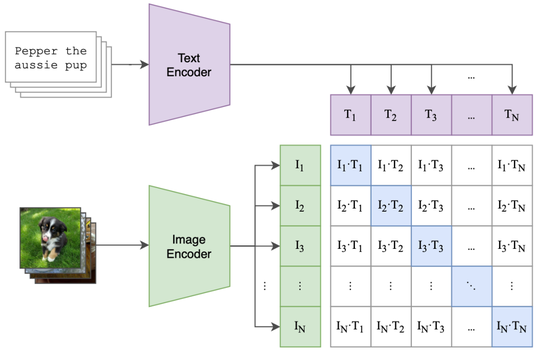
x: image

$$\Rightarrow P(x|y) = P(x, z_i|y) = P(x|z_i,y) \cdot P(z_i|y)$$

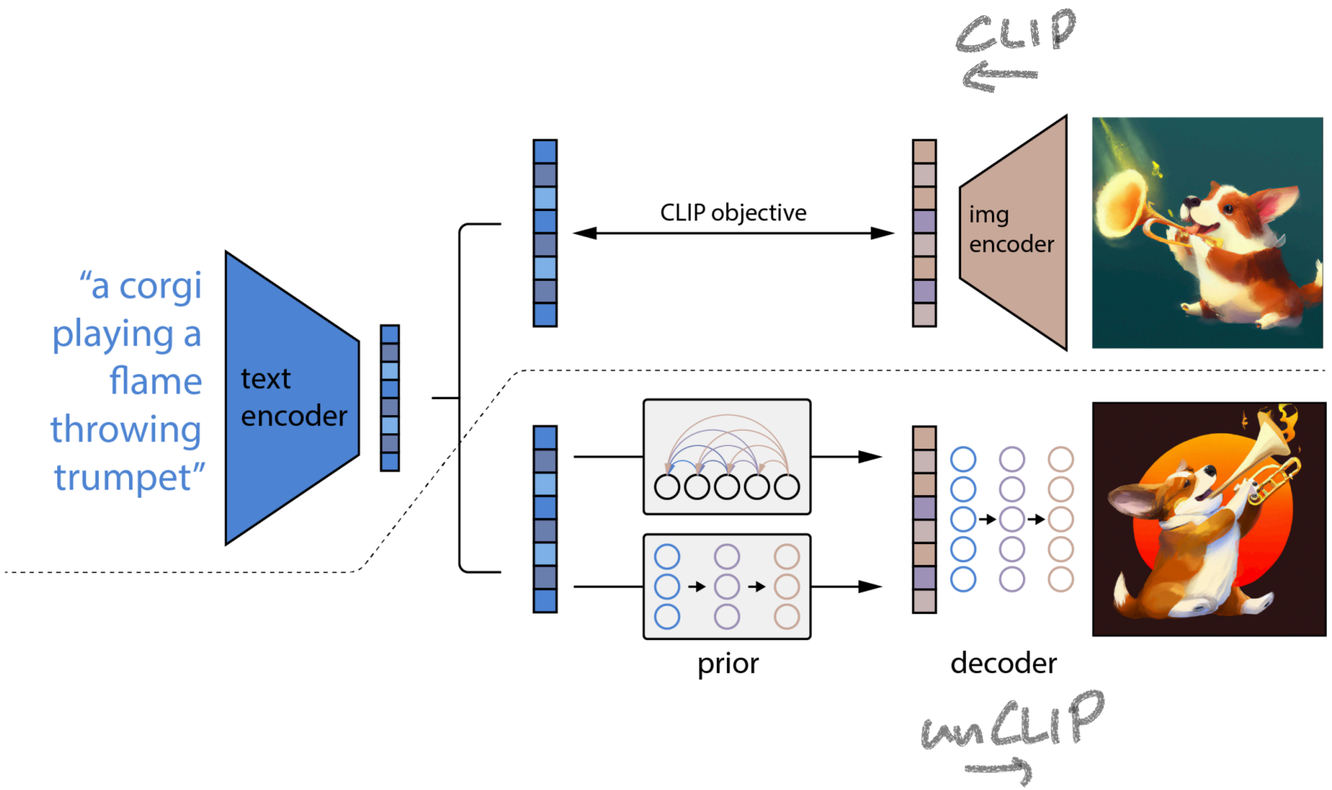
$z_i = f(x)$ deterministic

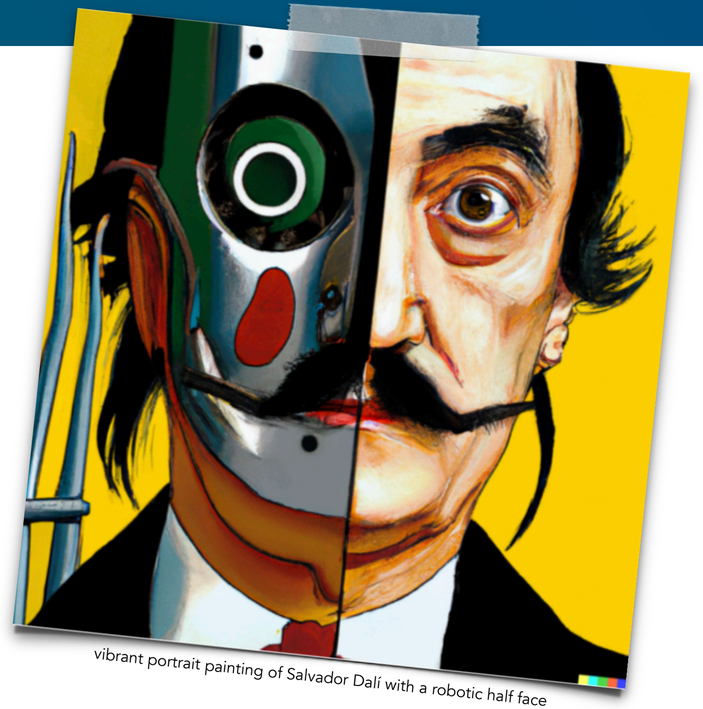
Definition of cond. prob

Excerpt from CLIP paper:

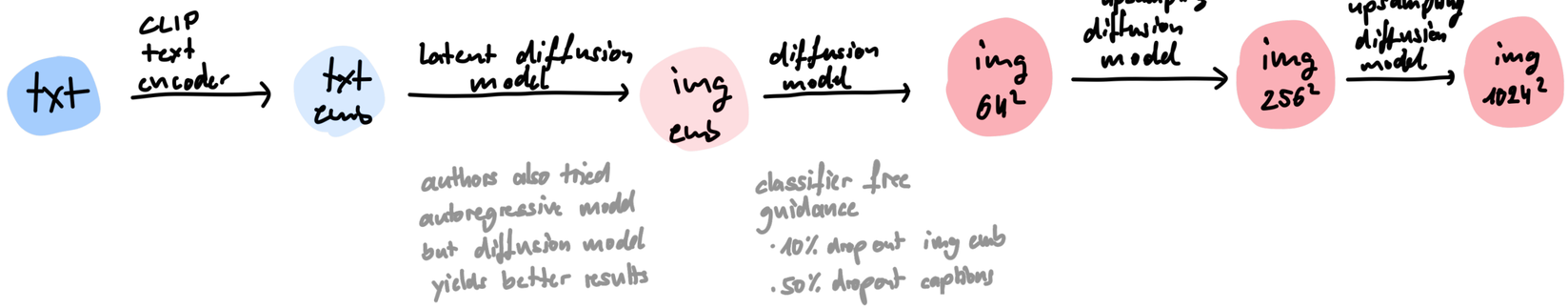


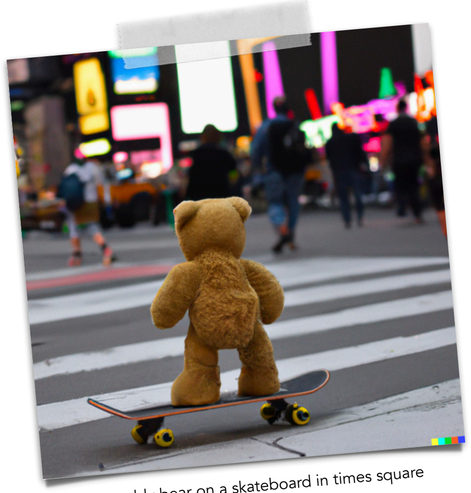
5.5 Improvements - Ramesh et al (DALL-E 2)





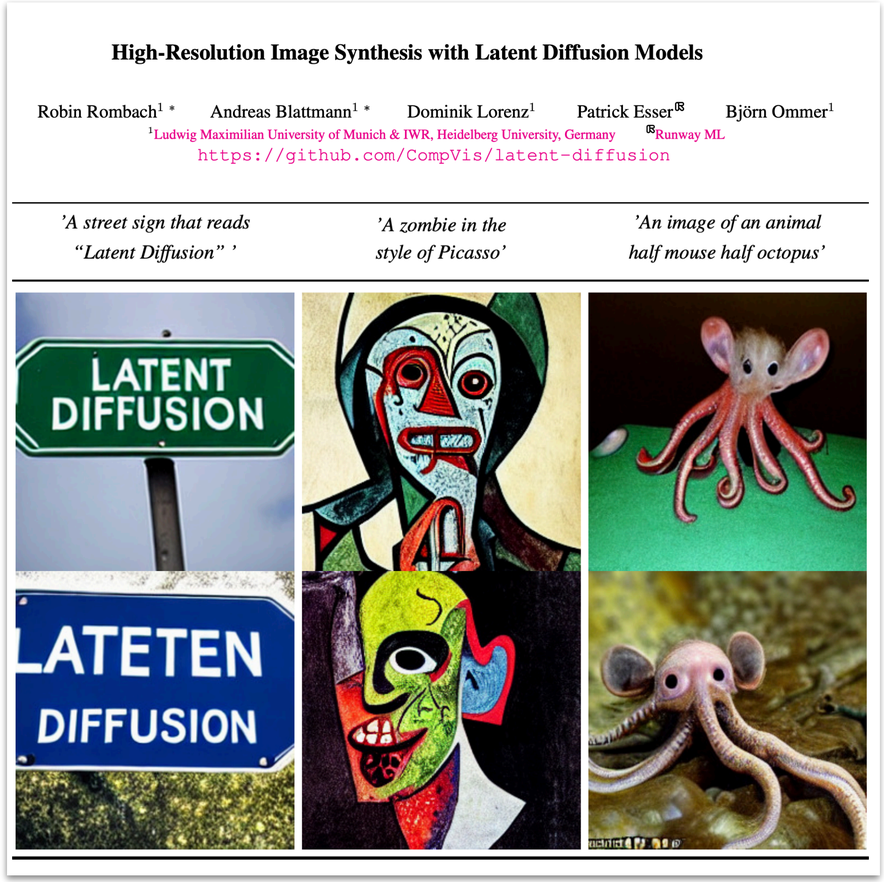
vibrant portrait painting of Salvador Dalí with a robotic half face





a teddy bear on a skateboard in times square

5.6 Improvements - Rombach et al (Stable Diffusion)



High-Resolution Image Synthesis with Latent Diffusion Models

Robin Rombach[1] * Andreas Blattmann[1] * Dominik Lorenz[1] Patrick Esser[R] Björn Ommer[1]
[1]Ludwig Maximilian University of Munich & IWR, Heidelberg University, Germany [R]Runway ML
https://github.com/CompVis/latent-diffusion

'A street sign that reads "Latent Diffusion" ' | *'A zombie in the style of Picasso'* | *'An image of an animal half mouse half octopus'*

seperatly trained VAE ($\mathcal{E} \circ \mathcal{D}$)

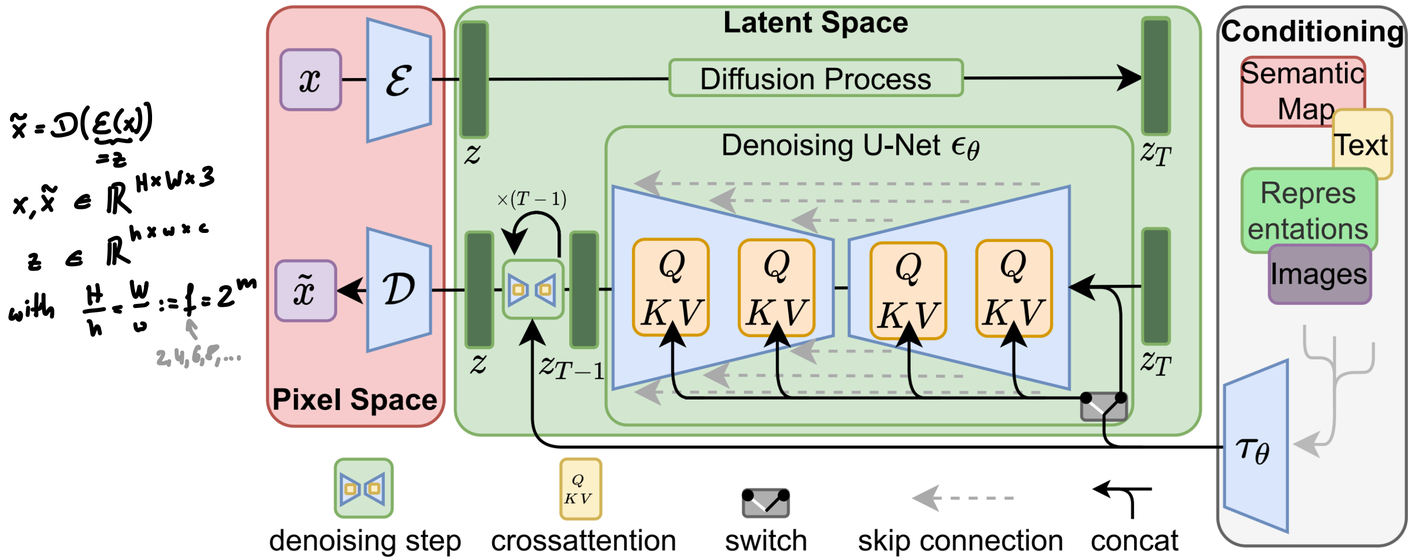
Denoising U-Net ϵ_θ

$\tilde{x} = \mathcal{D}(\underbrace{\mathcal{E}(x)}_{=z})$

$x, \tilde{x} \in \mathbb{R}^{H \times W \times 3}$

$z \in \mathbb{R}^{h \times w \times c}$

with $\frac{H}{h} = \frac{W}{w} := f = 2^m$ (2, 4, 6, 8, ...)



domain specific encoder τ_θ (trained jointly with ϵ_θ)

$$L_{LDM} = \left\langle \| \epsilon - \epsilon(z_t, t, \tau_\theta(y)) \|^2 \right\rangle_{\mathcal{E}(x), y, \epsilon \sim \mathcal{N}(0,1), t}$$

Main contributions:

1. Reduction of comp. requiremtnts through latent diffusion.
2. Cross-attention layer for general purpose (multi model) conditioning.

5.6 Improvements - Rombach et al (Stable Diffusion)

About the conditioning mechanism…

Domain specific encoder:

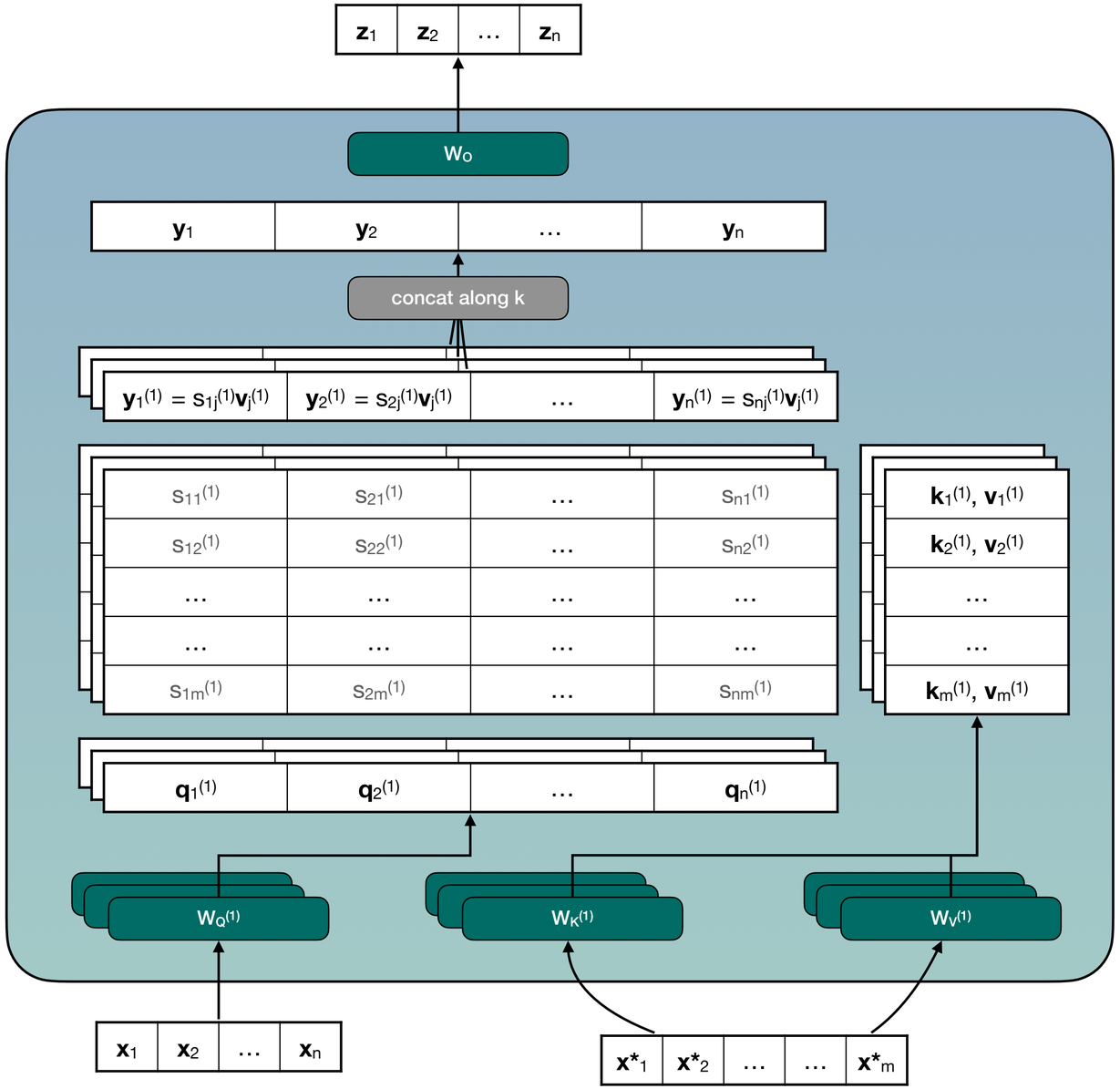
$$\tau_\theta : y \mapsto \tau_\theta(y) \in \mathbb{R}^{M \times d_\tau}$$

Cross-attention mechanism:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right) \cdot V$$

$$Q = W_Q^{(i)} \cdot \varphi_i(z_t) \qquad K = W_K^{(i)} \cdot \tau_\theta(y) \qquad V = W_V^{(i)} \cdot \tau_\theta(y)$$

$\varphi_i(z_t) \in \mathbb{R}^{N \times d_\epsilon^i}$ flattened intermediate representation of the UNet.



flattened rep UNet; N x dphii

domain specific decoder; M x dtau

6. UNet Architecture in Depth

...example implementation from Stable Diffusion repo...

```
unet.time_embed
✓ 0.0s                                                    Python
```

```
Sequential(
  (0): Linear(in_features=320, out_features=1280, bias=True)
  (1): SiLU()
  (2): Linear(in_features=1280, out_features=1280, bias=True)
)
```

```python
layer_names = []

for i, sequential_module in enumerate(unet.input_blocks):
    print(i, sequential_module._get_name())
    for j, layer in enumerate(sequential_module):
        print(f"  {i}.{j}", layer._get_name())
        layer_names.append(layer._get_name())

print()
for i, module in enumerate(unet.middle_block):
    layer_name = layer._get_name()
    print(f"  {i}.{j}", layer_name)
    layer_names.append(layer_name)

print()
for i, sequential_module in enumerate(unet.output_blocks):
    print(i, sequential_module._get_name())
    for j, layer in enumerate(sequential_module):
        print(f"  {i}.{j}", layer._get_name())
        layer_names.append(layer._get_name())

print(set(layer_names))
```

```
✓ 0.0s                                                    Python
```

```
0 TimestepEmbedSequential
  0.0 Conv2d
1 TimestepEmbedSequential
  1.0 ResBlock
  1.1 SpatialTransformer
2 TimestepEmbedSequential
  2.0 ResBlock
  2.1 SpatialTransformer
3 TimestepEmbedSequential
  3.0 Downsample
4 TimestepEmbedSequential
  4.0 ResBlock
  4.1 SpatialTransformer
5 TimestepEmbedSequential
  5.0 ResBlock
  5.1 SpatialTransformer
6 TimestepEmbedSequential
  6.0 Downsample
7 TimestepEmbedSequential
  7.0 ResBlock
  7.1 SpatialTransformer
8 TimestepEmbedSequential
  8.0 ResBlock
  8.1 SpatialTransformer
9 TimestepEmbedSequential
  9.0 Downsample
10 TimestepEmbedSequential
  10.0 ResBlock
11 TimestepEmbedSequential
  11.0 ResBlock

  0.0 ResBlock
  1.0 ResBlock
  2.0 ResBlock

0 TimestepEmbedSequential
  0.0 ResBlock
  1.0 ResBlock
1 TimestepEmbedSequential
  1.0 ResBlock
2 TimestepEmbedSequential
  2.0 ResBlock
  2.1 Upsample
3 TimestepEmbedSequential
  3.0 ResBlock
  3.1 SpatialTransformer
4 TimestepEmbedSequential
  4.0 ResBlock
  4.1 SpatialTransformer
5 TimestepEmbedSequential
  5.0 ResBlock
  5.1 SpatialTransformer
  5.2 Upsample
6 TimestepEmbedSequential
  6.0 ResBlock
  6.1 SpatialTransformer
7 TimestepEmbedSequential
  7.0 ResBlock
  7.1 SpatialTransformer
8 TimestepEmbedSequential
  8.0 ResBlock
  8.1 SpatialTransformer
  8.2 Upsample
9 TimestepEmbedSequential
  9.0 ResBlock
  9.1 SpatialTransformer
10 TimestepEmbedSequential
  10.0 ResBlock
  10.1 SpatialTransformer
11 TimestepEmbedSequential
  11.0 ResBlock
  11.1 SpatialTransformer
```

TimestepEmbedSequential:
Simply a sequential module that passes timestamp embeddings to the layers that support it.

```python
72
    Robin Rombach, 3 months ago | 1 author (Robin Rombach)
73  class TimestepEmbedSequential(nn.Sequential, TimestepBlock):
74      """
75      A sequential module that passes timestep embeddings to the children that
76      support it as an extra input.
77      """
78
79      def forward(self, x, emb, context=None):
80          for layer in self:
81              if isinstance(layer, TimestepBlock):
82                  x = layer(x, emb)
83              elif isinstance(layer, SpatialTransformer):
84                  x = layer(x, context)
85              else:
86                  x = layer(x)
87          return x
88
```

ldm/modules/diffusionmodules/openaimodels.py

The forward pass through the network:

```python
def forward(self, x, timesteps=None, context=None, y=None,**kwargs):
    """
    Apply the model to an input batch.
    :param x: an [N x C x ...] Tensor of inputs.
    :param timesteps: a 1-D batch of timesteps.
    :param context: conditioning plugged in via crossattn
    :param y: an [N] Tensor of labels, if class-conditional.
    :return: an [N x C x ...] Tensor of outputs.
    """
    assert (y is not None) == (
        self.num_classes is not None
    ), "must specify y if and only if the model is class-conditional"
    hs = []
    t_emb = timestep_embedding(timesteps, self.model_channels, repeat_only=False)
    emb = self.time_embed(t_emb)

    if self.num_classes is not None:
        assert y.shape[0] == x.shape[0]
        emb = emb + self.label_emb(y)

    h = x.type(self.dtype)
    for module in self.input_blocks:
        h = module(h, emb, context)
        hs.append(h)
    h = self.middle_block(h, emb, context)
    for module in self.output_blocks:
        h = th.cat([h, hs.pop()], dim=1)
        h = module(h, emb, context)
    h = h.type(x.dtype)
    if self.predict_codebook_ids:
        return self.id_predictor(h)
    else:
        return self.out(h)
```

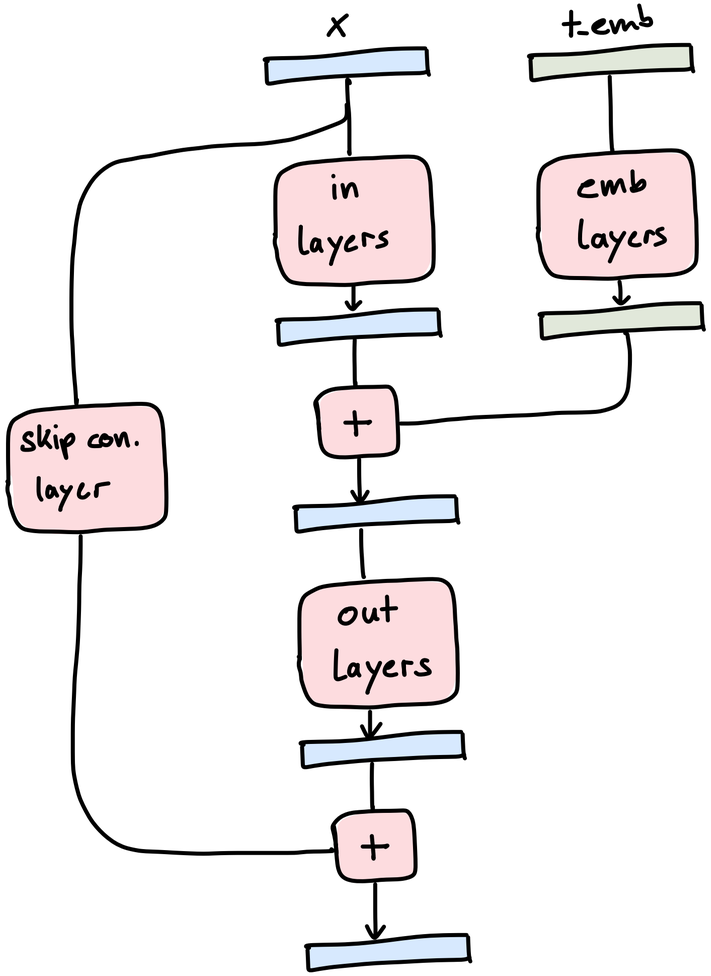
6. UNet Architecture in Depth

ResBlock

```
unet.input_blocks[1][0]
✓ 0.0s                                                    Python
ResBlock(
  (in_layers): Sequential(
    (0): GroupNorm32(32, 320, eps=1e-05, affine=True)
    (1): SiLU()
    (2): Conv2d(320, 320, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  )
  (h_upd): Identity()
  (x_upd): Identity()
  (emb_layers): Sequential(
    (0): SiLU()
    (1): Linear(in_features=1280, out_features=320, bias=True)
  )
  (out_layers): Sequential(
    (0): GroupNorm32(32, 320, eps=1e-05, affine=True)
    (1): SiLU()
    (2): Dropout(p=0, inplace=False)
    (3): Conv2d(320, 320, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  )
  (skip_connection): Identity()
)
```

```
161
      Robin Rombach, 3 months ago | 1 author (Robin Rombach)
162   class ResBlock(TimestepBlock):
163       """
164       A residual block that can optionally change the number of channels.

253                                   ...
254       def _forward(self, x, emb):
255           if self.updown:
256               in_rest, in_conv = self.in_layers[:-1], self.in_layers[-1]
257               h = in_rest(x)
258               h = self.h_upd(h)
259               x = self.x_upd(x)
260               h = in_conv(h)
261           else:
262               h = self.in_layers(x)
263           emb_out = self.emb_layers(emb).type(h.dtype)
264           while len(emb_out.shape) < len(h.shape):
265               emb_out = emb_out[..., None]
266           if self.use_scale_shift_norm:
267               out_norm, out_rest = self.out_layers[0], self.out_layers[1:]
268               scale, shift = th.chunk(emb_out, 2, dim=1)
269               h = out_norm(h) * (1 + scale) + shift
270               h = out_rest(h)
271           else:
272               h = h + emb_out
273               h = self.out_layers(h)
274           return self.skip_connection(x) + h
275
```

ldm/modules/diffusionmodules/openaimodels.py



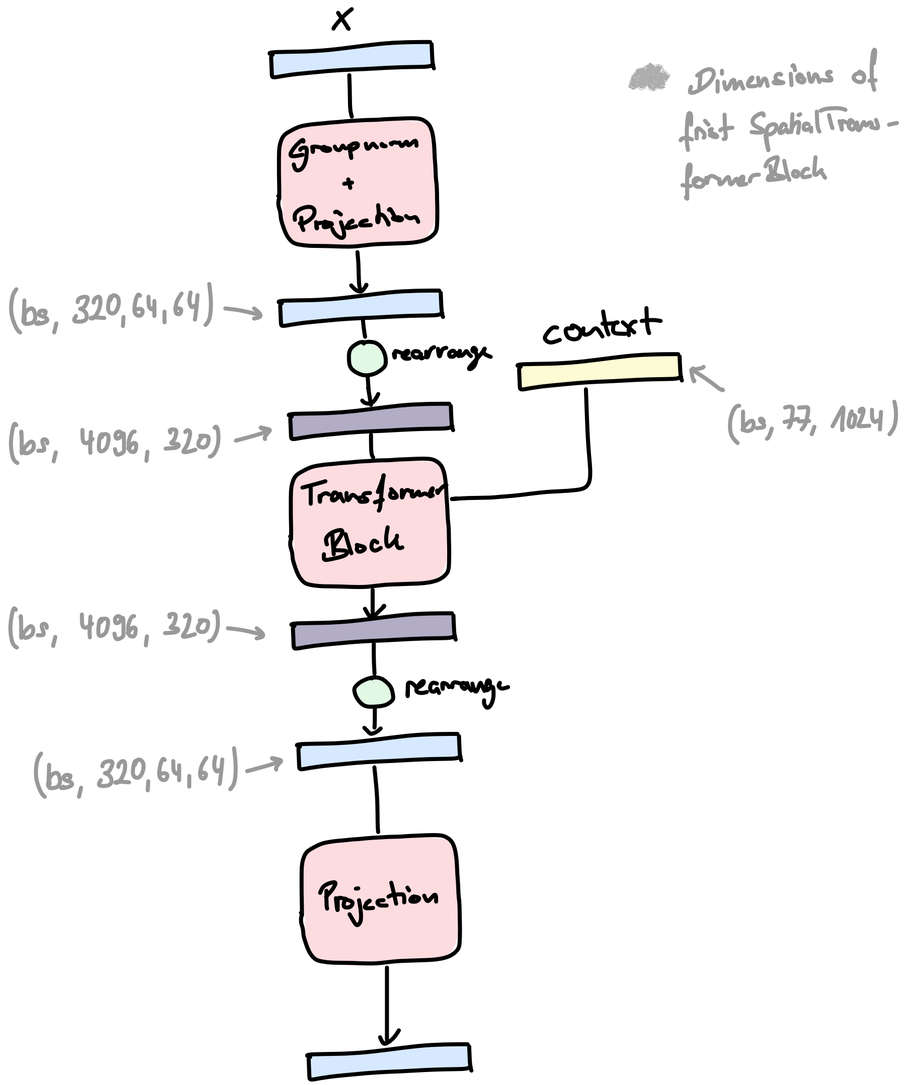
6. UNet Architecture in Depth

Spatial Transformer (Part 1)

```
unet.input_blocks[1][1]
✓ 0.0s                                                    Python
SpatialTransformer(
  (norm): GroupNorm(32, 320, eps=1e-06, affine=True)
  (proj_in): Linear(in_features=320, out_features=320, bias=True)
  (transformer_blocks): ModuleList(
    (0): BasicTransformerBlock(
      (attn1): CrossAttention(
        (to_q): Linear(in_features=320, out_features=320, bias=False)
        (to_k): Linear(in_features=320, out_features=320, bias=False)
        (to_v): Linear(in_features=320, out_features=320, bias=False)
        (to_out): Sequential(
          (0): Linear(in_features=320, out_features=320, bias=True)
          (1): Dropout(p=0.0, inplace=False)
        )
      )
      (ff): FeedForward(
        (net): Sequential(
          (0): GEGLU(
            (proj): Linear(in_features=320, out_features=2560, bias=True)
          )
          (1): Dropout(p=0.0, inplace=False)
          (2): Linear(in_features=1280, out_features=320, bias=True)
        )
      )
      (attn2): CrossAttention(
        (to_q): Linear(in_features=320, out_features=320, bias=False)
        (to_k): Linear(in_features=1024, out_features=320, bias=False)
        (to_v): Linear(in_features=1024, out_features=320, bias=False)
        (to_out): Sequential(
          (0): Linear(in_features=320, out_features=320, bias=True)
          (1): Dropout(p=0.0, inplace=False)
        )
      )
      (norm1): LayerNorm((320,), eps=1e-05, elementwise_affine=True)
      (norm2): LayerNorm((320,), eps=1e-05, elementwise_affine=True)
      (norm3): LayerNorm((320,), eps=1e-05, elementwise_affine=True)
    )
  )
  (proj_out): Linear(in_features=320, out_features=320, bias=True)
)
```

```python
class SpatialTransformer(nn.Module):
    """
    Transformer block for image-like data.
    First, project the input (aka embedding)
    and reshape to b, t, d.
    Then apply standard transformer action.
    Finally, reshape to image
    NEW: use_linear for more efficiency instead of the 1x1 convs
    """
    ...
    def forward(self, x, context=None):
        # note: if no context is given, cross-attention defaults to self-attention
        if not isinstance(context, list):
            context = [context]
        b, c, h, w = x.shape
        x_in = x
        x = self.norm(x)
        if not self.use_linear:
            x = self.proj_in(x)
        x = rearrange(x, 'b c h w -> b (h w) c').contiguous()
        if self.use_linear:
            x = self.proj_in(x)
        for i, block in enumerate(self.transformer_blocks):
            x = block(x, context=context[i])
        if self.use_linear:
            x = self.proj_out(x)
        x = rearrange(x, 'b (h w) c -> b c h w', h=h, w=w).contiguous()
        if not self.use_linear:
            x = self.proj_out(x)
        return x + x_in
```

ldm/modules/attention.py



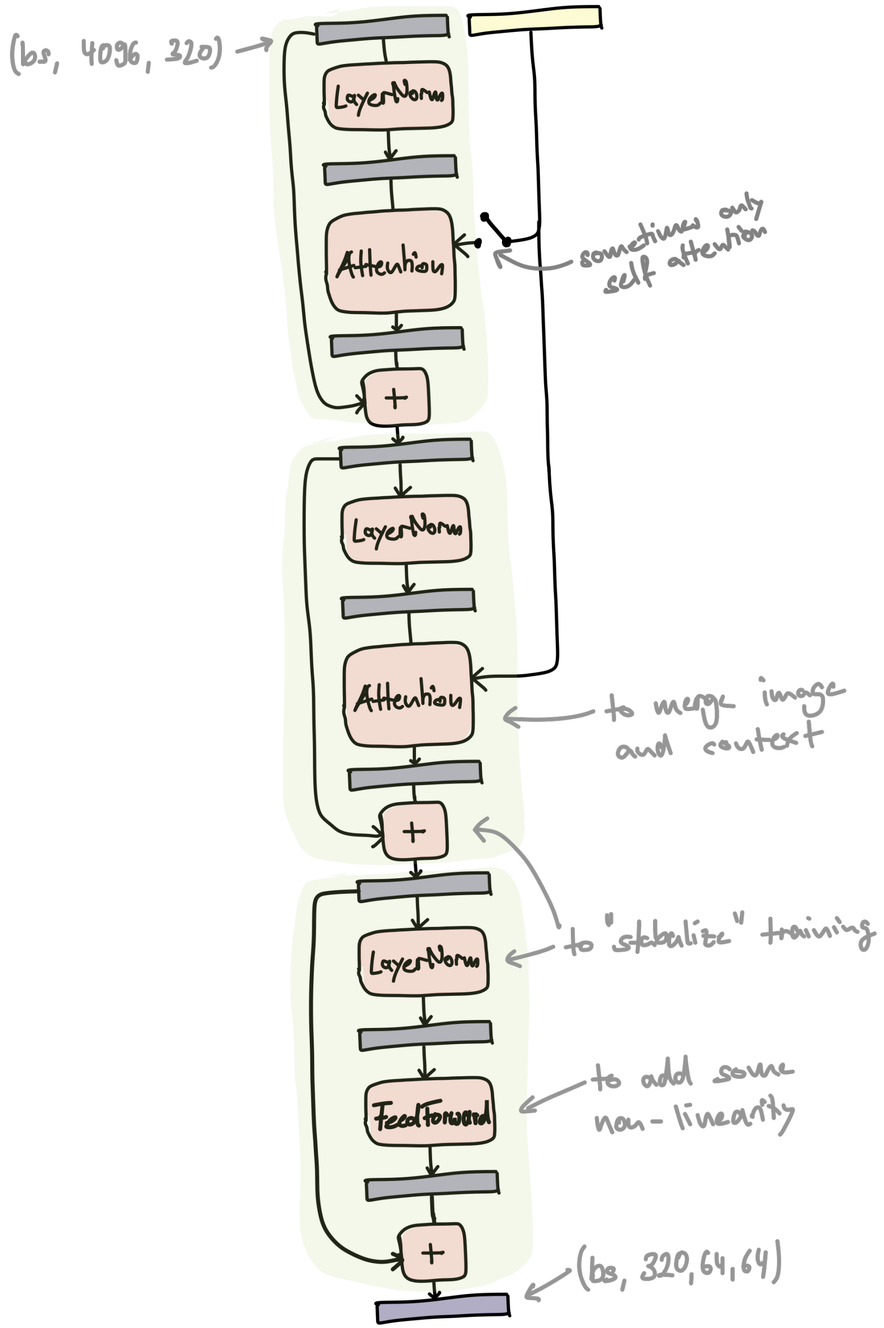
6. UNet Architecture in Depth

Spatial Transformer (Part 2)

```
unet.input_blocks[1][1]
✓ 0.0s                                                    Python

SpatialTransformer(
  (norm): GroupNorm(32, 320, eps=1e-06, affine=True)
  (proj_in): Linear(in_features=320, out_features=320, bias=True)
  (transformer_blocks): ModuleList(
    (0): BasicTransformerBlock(
      (attn1): CrossAttention(
        (to_q): Linear(in_features=320, out_features=320, bias=False)
        (to_k): Linear(in_features=320, out_features=320, bias=False)
        (to_v): Linear(in_features=320, out_features=320, bias=False)
        (to_out): Sequential(
          (0): Linear(in_features=320, out_features=320, bias=True)
          (1): Dropout(p=0.0, inplace=False)
        )
      )
      (ff): FeedForward(
        (net): Sequential(
          (0): GEGLU(
            (proj): Linear(in_features=320, out_features=2560, bias=True)
          )
          (1): Dropout(p=0.0, inplace=False)
          (2): Linear(in_features=1280, out_features=320, bias=True)
        )
      )
      (attn2): CrossAttention(
        (to_q): Linear(in_features=320, out_features=320, bias=False)
        (to_k): Linear(in_features=1024, out_features=320, bias=False)
        (to_v): Linear(in_features=1024, out_features=320, bias=False)
        (to_out): Sequential(
          (0): Linear(in_features=320, out_features=320, bias=True)
          (1): Dropout(p=0.0, inplace=False)
        )
      )
      (norm1): LayerNorm((320,), eps=1e-05, elementwise_affine=True)
      (norm2): LayerNorm((320,), eps=1e-05, elementwise_affine=True)
      (norm3): LayerNorm((320,), eps=1e-05, elementwise_affine=True)
    )
  )
  (proj_out): Linear(in_features=320, out_features=320, bias=True)
)
```

```
Robin Rombach, 3 months ago | 1 author (Robin Rombach)
246  class BasicTransformerBlock(nn.Module):      Robin Rombach, 3 months ago • release more models

271      def _forward(self, x, context=None):
272          x = self.attn1(self.norm1(x), context=context if self.disable_self_attn else None) + x
273          x = self.attn2(self.norm2(x), context=context) + x
274          x = self.ff(self.norm3(x)) + x
275          return x
276
```

ldm/modules/attention.py



6. UNet Architecture in Depth

Spatial Transformer (Part 3)

```
unet.input_blocks[1][1]
✓ 0.0s                                                    Python
SpatialTransformer(
  (norm): GroupNorm(32, 320, eps=1e-06, affine=True)
  (proj_in): Linear(in_features=320, out_features=320, bias=True)
  (transformer_blocks): ModuleList(
    (0): BasicTransformerBlock(
      (attn1): CrossAttention(
        (to_q): Linear(in_features=320, out_features=320, bias=False)
        (to_k): Linear(in_features=320, out_features=320, bias=False)
        (to_v): Linear(in_features=320, out_features=320, bias=False)
        (to_out): Sequential(
          (0): Linear(in_features=320, out_features=320, bias=True)
          (1): Dropout(p=0.0, inplace=False)
        )
      )
      (ff): FeedForward(
        (net): Sequential(
          (0): GEGLU(
            (proj): Linear(in_features=320, out_features=2560, bias=True)
          )
          (1): Dropout(p=0.0, inplace=False)
          (2): Linear(in_features=1280, out_features=320, bias=True)
        )
      )
      (attn2): CrossAttention(
        (to_q): Linear(in_features=320, out_features=320, bias=False)
        (to_k): Linear(in_features=1024, out_features=320, bias=False)
        (to_v): Linear(in_features=1024, out_features=320, bias=False)
        (to_out): Sequential(
          (0): Linear(in_features=320, out_features=320, bias=True)
          (1): Dropout(p=0.0, inplace=False)
        )
      )
      (norm1): LayerNorm((320,), eps=1e-05, elementwise_affine=True)
      (norm2): LayerNorm((320,), eps=1e-05, elementwise_affine=True)
      (norm3): LayerNorm((320,), eps=1e-05, elementwise_affine=True)
    )
  )
  (proj_out): Linear(in_features=320, out_features=320, bias=True)
)
```

```
144
    Dango233, 2 months ago | 2 authors (Robin Rombach and others)
145 class CrossAttention(nn.Module):
146     def __init__(self, query_dim, context_dim=None, heads=8, dim_head=64, dropout=0.):
147         super().__init__()
148         inner_dim = dim_head * heads
149         context_dim = default(context_dim, query_dim)
150
151         self.scale = dim_head ** -0.5
152         self.heads = heads
153
154         self.to_q = nn.Linear(query_dim, inner_dim, bias=False)
155         self.to_k = nn.Linear(context_dim, inner_dim, bias=False)
156         self.to_v = nn.Linear(context_dim, inner_dim, bias=False)
157
158         self.to_out = nn.Sequential(
159             nn.Linear(inner_dim, query_dim),
160             nn.Dropout(dropout)
161         )      Robin Rombach, 3 months ago • release more models
162
163     def forward(self, x, context=None, mask=None):
164         h = self.heads
165
166         q = self.to_q(x)
167         context = default(context, x)
168         k = self.to_k(context)
169         v = self.to_v(context)
170
171         q, k, v = map(lambda t: rearrange(t, 'b n (h d) -> (b h) n d', h=h), (q, k, v))
172
173         # force cast to fp32 to avoid overflowing
174         if _ATTN_PRECISION =="fp32":
175             with torch.autocast(enabled=False, device_type = 'cuda'):
176                 q, k = q.float(), k.float()
177                 sim = einsum('b i d, b j d -> b i j', q, k) * self.scale
178         else:
179             sim = einsum('b i d, b j d -> b i j', q, k) * self.scale
180
181         del q, k
182
183         if exists(mask):
184             mask = rearrange(mask, 'b ... -> b (...)')
185             max_neg_value = -torch.finfo(sim.dtype).max
186             mask = repeat(mask, 'b j -> (b h) () j', h=h)
187             sim.masked_fill_(~mask, max_neg_value)
188
189         # attention, what we cannot get enough of
190         sim = sim.softmax(dim=-1)
191
192         out = einsum('b i j, b j d -> b i d', sim, v)
193         out = rearrange(out, '(b h) n d -> b n (h d)', h=h)
194         return self.to_out(out)
195
```

```
def default(val, d):
    if exists(val):
        return val
    return d() if isfunction(d) else d
```

example dimensions of first CrossAttention Block:

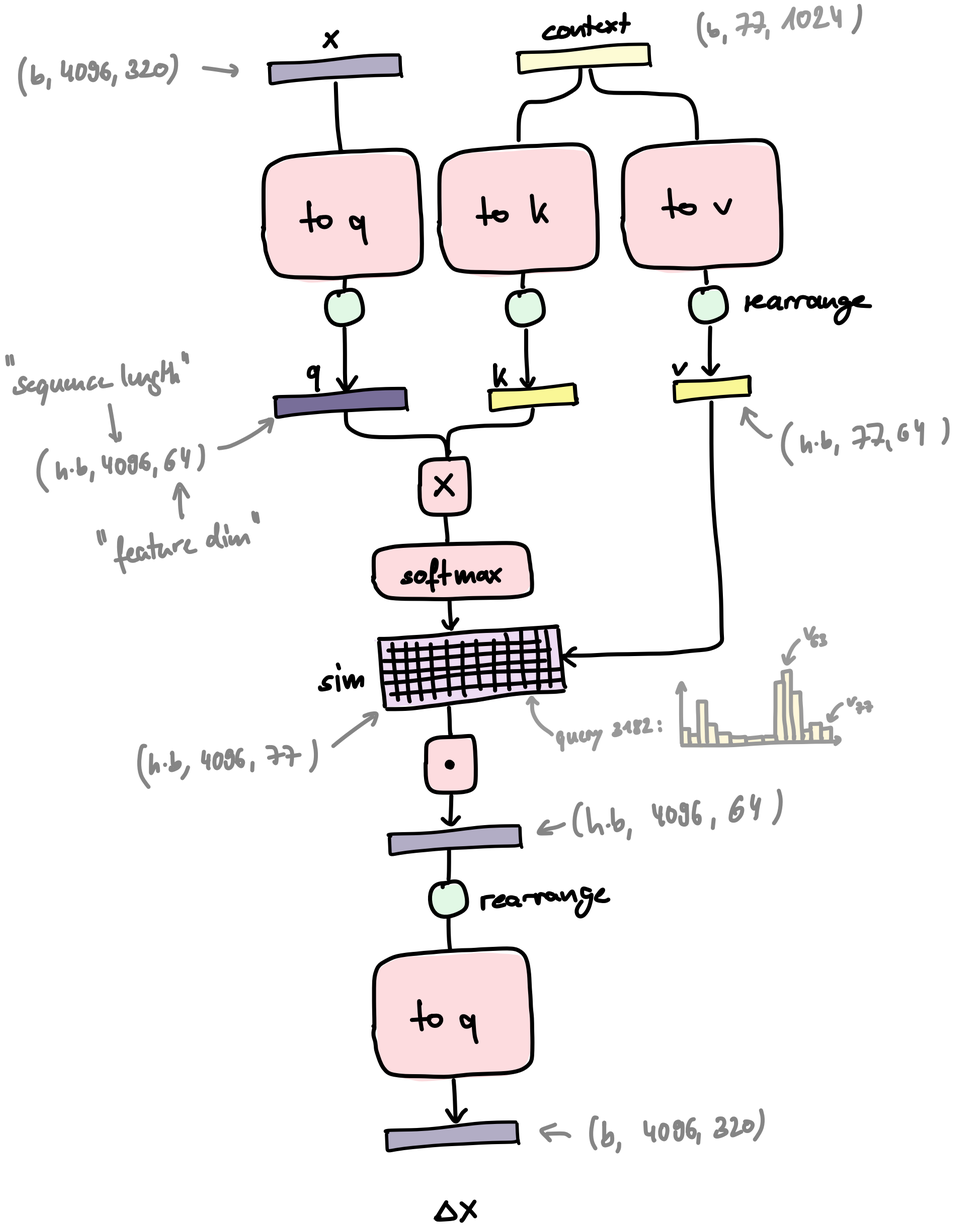
```
x = torch.Size([1*bs, 4096, 320])
context = torch.Size([1*bs, 77, 1024])
h = 5
q = torch.Size([1*bs, 4096, 320])
k = torch.Size([1*bs, 77, 320])
v = torch.Size([1*bs, 77, 320])

q = torch.Size([5*bs, 4096, 64])
k = torch.Size([5*bs, 77, 64])
v = torch.Size([5*bs, 77, 64])

sim = torch.Size([5*bs, 4096, 77])

out = torch.Size([5*bs, 4096, 64])

out = torch.Size([1*bs, 4096, 320])

ret = torch.Size([1*bs, 4096, 320])
```

ldm/modules/attention.py

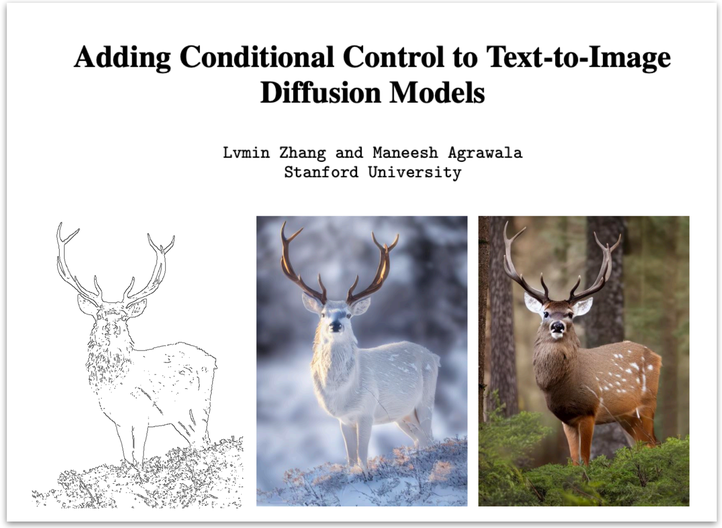
6. UNet Architecture in Depth

Spatial Transformer (Part 3)

```
unet.input_blocks[1][1]
✓ 0.0s                                                    Python
SpatialTransformer(
  (norm): GroupNorm(32, 320, eps=1e-06, affine=True)
  (proj_in): Linear(in_features=320, out_features=320, bias=True)
  (transformer_blocks): ModuleList(
    (0): BasicTransformerBlock(
      (attn1): CrossAttention(
        (to_q): Linear(in_features=320, out_features=320, bias=False)
        (to_k): Linear(in_features=320, out_features=320, bias=False)
        (to_v): Linear(in_features=320, out_features=320, bias=False)
        (to_out): Sequential(
          (0): Linear(in_features=320, out_features=320, bias=True)
          (1): Dropout(p=0.0, inplace=False)
        )
      )
      (ff): FeedForward(
        (net): Sequential(
          (0): GEGLU(
            (proj): Linear(in_features=320, out_features=2560, bias=True)
          )
          (1): Dropout(p=0.0, inplace=False)
          (2): Linear(in_features=1280, out_features=320, bias=True)
        )
      )
      (attn2): CrossAttention(
        (to_q): Linear(in_features=320, out_features=320, bias=False)
        (to_k): Linear(in_features=1024, out_features=320, bias=False)
        (to_v): Linear(in_features=1024, out_features=320, bias=False)
        (to_out): Sequential(
          (0): Linear(in_features=320, out_features=320, bias=True)
          (1): Dropout(p=0.0, inplace=False)
        )
      )
      (norm1): LayerNorm((320,), eps=1e-05, elementwise_affine=True)
      (norm2): LayerNorm((320,), eps=1e-05, elementwise_affine=True)
      (norm3): LayerNorm((320,), eps=1e-05, elementwise_affine=True)
    )
  )
  (proj_out): Linear(in_features=320, out_features=320, bias=True)
)
```



7. ControlNet (Feb. 2023)



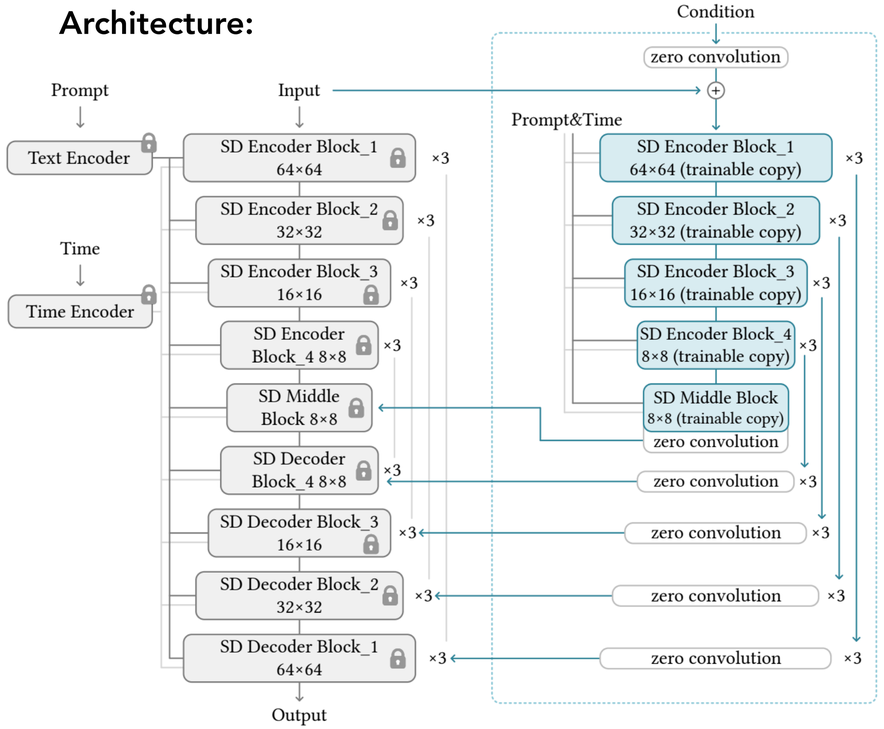
Main contributions:

1. New method to of conditioning large text2image diffusion models through *"zero convolutions"*.

Improvement over fine-tuning:

- Network always keeps production ready state, i.e., it does not forget its general txt2image capabilities.
- Training is as fast as fine-tuning.
- No overfitting.

Architecture:

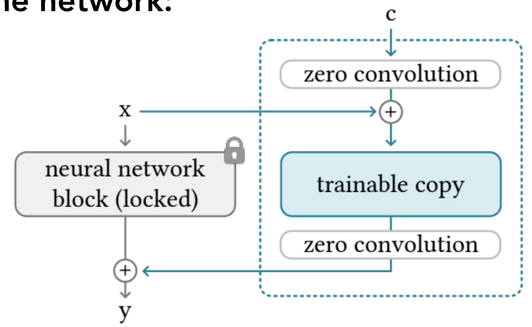


Normal frozen SD

ControlNet

- exact copy of SD layers and weights
- features are merged by zero-convolution layers

Detailed view of one "block" somewhere in the network:



Zero convolutions: Convolutional layers that are initialised with vanishing weights and biases (W=0, B=0).

- At the first training step: ControlNet has no effect.
- After optimisation: ControlNet effectively augments each feature vector according to the conditioning vector.