Diffusion models, a short tutorial

Nicolas Cherel

June 6th 2023

Table of contents

Introduction

Motivations

Diffusion models belong to the family of generative models: GANs, VAEs, normalizing flows, etc.

Goal: Learn to sample/generate new data points from an unknown data distribution.

Motivations

Unconditional sampling

Conditional sampling

Condition = class, text

Motivations

Unconditional sampling

Conditional sampling

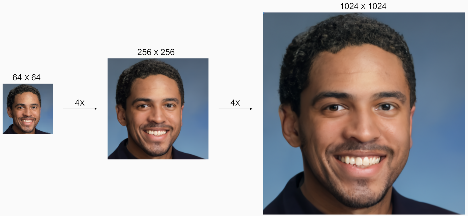
Condition $=$ observations (inverse problems)

Why diffusion?

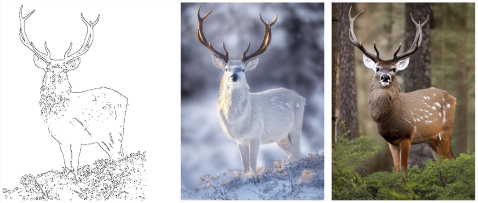
Compared to other generative models, diffusion:

+ produces high-quality and diverse samples
+ has no problem of mode collapse
+ is easy to train
- is slow
- has no latent space

Why diffusion?



Super-Resolution (SR3)



Controlled synthesis (ControlNet)

Uncropping (Palette)

Theory

Disclaimer

Mainly about diffusion as described in Ho, Jain, and Abbeel, *Denoising Diffusion Probabilistic Models*; based on a Markov model:

$$q(\mathrm{x}_t \mid \mathrm{x}_{t-1}) = \mathcal{N}\left(\sqrt{1-\beta_t}\mathrm{x}_{t-1}, \beta_t \boldsymbol{I}\right)$$

✖ Not about score-based approaches using stochastic differential equations:

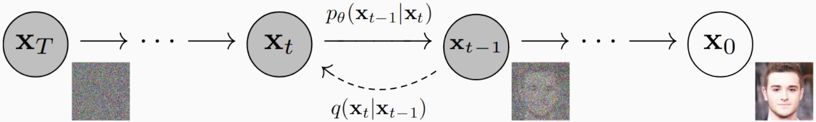
$$\mathrm{dx} = -\frac{1}{2}\beta(t)\mathrm{x}\,\mathrm{d}t + \sqrt{\beta(t)}\mathrm{dw}$$

References
CVPR 2022 tutorial on diffusion models:
`https://cvpr2022-tutorial-diffusion-models.github.io/`

Disclaimer

Interrupt for questions if needed

High-level overview



The famous image-to-noise and noise-to-image diagram

- We don't know how to sample from $q(\mathrm{x}_0)$
- We know how to sample from $q(\mathrm{x}_T)$
- We know how to go from x_0 to x_T
- We learn how to go from x_T to x_0

Forward process

Let's introduce $q(\mathrm{x}_0)$ the data distribution of images. We define the forward process for t ranging from 1 to T, defining the random variables $q(\mathrm{x}_t)$:

$$\mathrm{x}_0 \longrightarrow \cdots \longrightarrow \mathrm{x}_{t-1} \longrightarrow \mathrm{x}_t \longrightarrow \cdots \longrightarrow \mathrm{x}_T$$

$$q(\mathrm{x}_t \mid \mathrm{x}_{t-1}) = \mathcal{N}\left(\sqrt{1-\beta_t}\mathrm{x}_{t-1}, \beta_t \boldsymbol{I}\right)$$

β_t are small (< 0.02) and increasing slowly, T is large (1000 usually).

Forward process

Objective: $q\left(\mathrm{x}_T \mid \mathrm{x}_0\right) \approx \mathcal{N}(0, \boldsymbol{I})$

At step t we have:

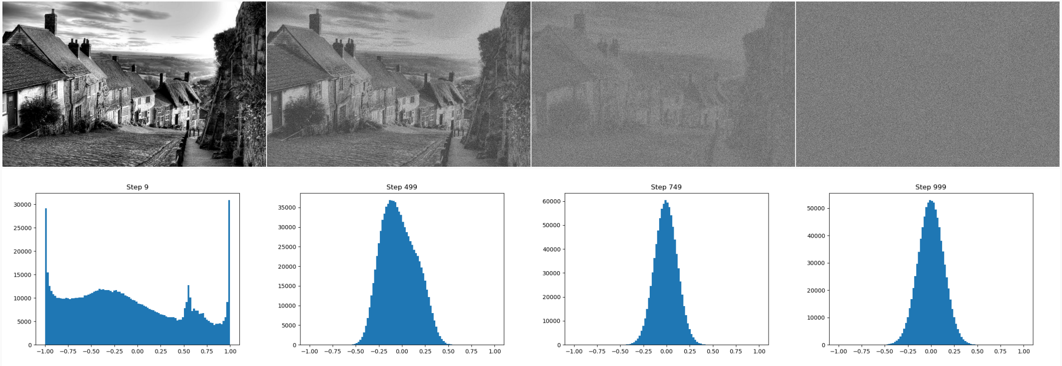
$$q\left(\mathrm{x}_t \mid x_0\right) = \mathcal{N}\left(\sqrt{\prod_{i=1}^{t}(1-\beta_i)}\mathrm{x}_0, \left(1-\prod_{i=1}^{t}(1-\beta_i)\right)\boldsymbol{I}\right)$$

Setting $\beta_1 = 0.0001$ and linearly increasing to $\beta_T = 0.02$, we get:

$$\mu = 0.0063 \cdot x_0 \quad \Sigma = 0.99996 \cdot \boldsymbol{I}$$

We consider the β parameters to be fixed but they could be learned as well.

Forward process



Convergence to a normal distribution

Preprocessing: Normalize data to be in $[-1, 1]$

Backward process

We want to learn the reverse processing, knowing that $p_\theta(x_t \mid x_{t+1})$ is Gaussian, but of unknown mean and variance:

$$x_t \longleftarrow x_{t+1}$$

$$p_\theta(x_t \mid x_{t+1}) = \mathcal{N}(\mu_\theta(x_{t+1}, t), \Sigma_\theta(x_{t+1}, t))$$

The parameters of the Gaussian are predicted by a neural network from x_{t+1}.

Ho et al. only predict the mean with a fixed variance schedule.

$$p_\theta(x_t \mid x_{t+1}) = \mathcal{N}(\mu_\theta(x_{t+1}, t), \sigma_t^2 \boldsymbol{I})$$

Loss function

Rewriting the log-likelihood lower bound[1], the loss is mostly the Kullback-Leibler divergence between 2 Gaussians for each timestep t:

$$
\begin{aligned}
\mathcal{L} &= \sum_{t=1}^{T} D_{\mathrm{KL}}\left(q(x_{t-1} \mid x_t, x_0) || p_\theta(x_{t-1} \mid x_t)\right) \\
&= \sum_{t=1}^{T} D_{\mathrm{KL}}\left(\mathcal{N}\left(\mu_t(x_t, x_0), \Sigma_t(x_t, x_0)\right) || \mathcal{N}(\mu_\theta(x_t, t), \sigma_t^2 \boldsymbol{I})\right) \\
&= \sum_{t=1}^{T} w(t) \|\mu_t(x_t, x_0) - \mu_\theta(x_t, t)\|^2 + C
\end{aligned}
$$

[1] quite long derivations

Parametrizations

We want our network to minimize $\|\mu_t(x_t, x_0) - \mu_\theta(x_t, t)\|^2$. We have different options for the output of the neural network by rewriting the μ_t as a function of x_t, x_0, and ϵ (noise added to x_0 to get x_t):

$$\underbrace{\mu_t(x_t, x_0)}_{1} = \underbrace{a_t x_0 + b_t x_t}_{2} = \underbrace{c_t x_t + d_t \epsilon}_{3}$$

1. Predict μ
2. Predict x_0, original clean image
3. Predict ϵ, residual noise

The parametrization changes the weighting term $w(t)$ in the sum.

Inference: sampling from the distribution

We start from noise $x_T \sim \mathcal{N}(0, \boldsymbol{I})$ and go backward in the Markov Chain, using the predicted mean by the network:

$$x_t \sim p_\theta(x_t \mid x_{t+1}) = \mathcal{N}(\mu_\theta(x_{t+1}, t), \sigma_t^2 \boldsymbol{I})$$

At each inference step, we sample from a Gaussian. We need to go through the networks T times, which is a lot.

Practical details

Practical details

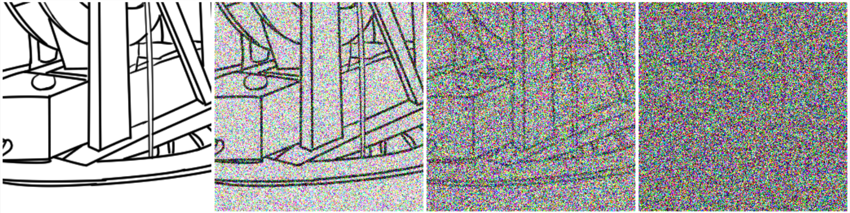
Practical considerations when working on diffusion, partially based on my experiments, partially from papers, githubs, etc.

⚠ Some of these "truths" may only hold in my special case (inpainting)

Network

For diffusion, it is common to use a single network for all timesteps. Something UNet-like, which is very common in denoising and image-to-image problems

- with enough parameters
- with time information



Training

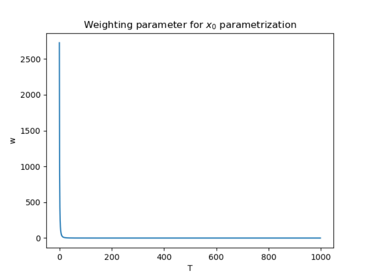
Using the x_0-parametrization, the training loop is the following:

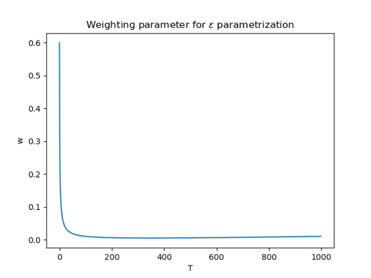
```
for images in train_dataloader:
    t = torch.random.randint(1, 1000, shape=(batch_size,1))
    noise = torch.randn_like(images)

    x_t = torch.sqrt(alpha[t]) * images + torch.sqrt(1 - alpha[t])
 * noise
    x_0 = model(x_t, t)

    loss = torch.mean(weight(t) * mse_loss(x_0, images))
```

Weighting term

Theoretical loss function has a weighting term, which depends on the parametrization:

$$\mathcal{L} = \sum_{t=1}^{T} \boldsymbol{w}(\boldsymbol{t}) \|\mu_t - \mu_\theta(x_t, t)\|^2$$





Idea: for small timesteps, weight more the error (task is easier). For large timesteps, loss is less important.

Weighting term

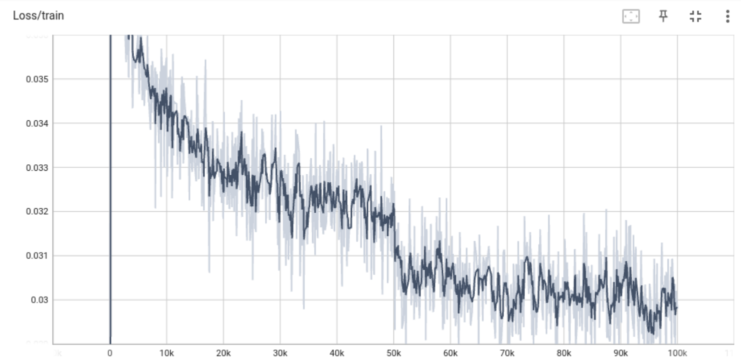
Ho et al. present their simple loss $w_\epsilon(t) = 1$
In my experiments, $w_{x_0}(t) = 1$ works well (stable) but underweights the early steps, which makes the outputs too smooth.

Advice

In any case, gradient is going to be very noisy. Avoid small batches and use different timesteps t

Training curve

Only one thing to monitor: L2 loss. Usually decreases monotically (up to statistical noise). Often, lower loss $\rightarrow$ better results visually.



Training curve for inpainting (easier than synthesis)

Time

Training time
Incremental improvements. Decent results early on, can keep going forever

Inference time
Depends on your network depth.

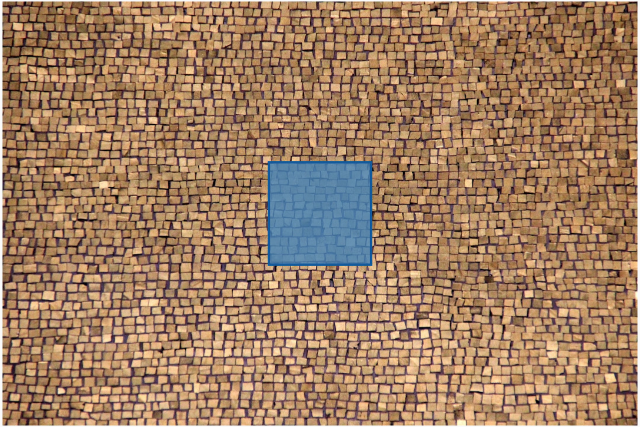
Small networks: < 2 seconds

Large networks: ~ 1.5 min

Application: Inpainting

Experiments

Inpainting with training on a single large image of texture.



Test set in blue

Setting

Goal: test the diffusion framework, and compare the approach to a direct inpainting problem

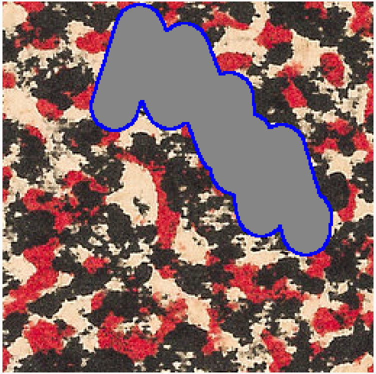
- Small UNet with 160k parameters
- Fast training in under 1 hour

Traditional inpainting

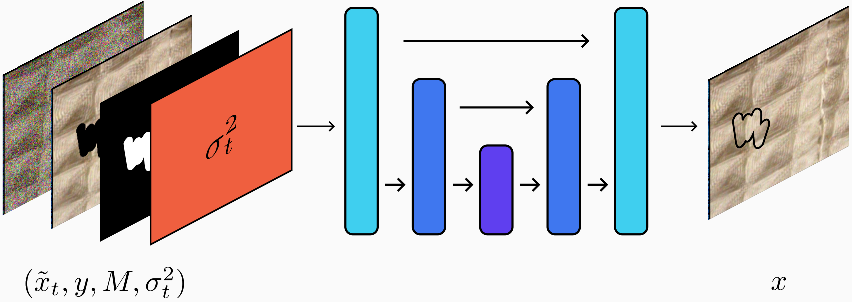
Inpainting with a simple reconstruction loss, aka **Regression**:

$$\mathcal{L} = \|x - f_\theta(x \circ (1 - M))\|^2$$

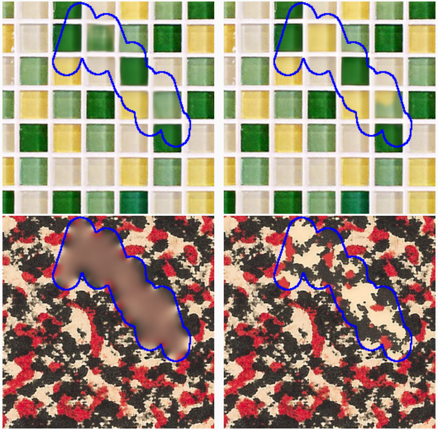
"Best" solution for this problem is very smooth (average of all possible solutions). Recover sharp edges with additional loss terms: perceptual loss, GAN loss, etc.



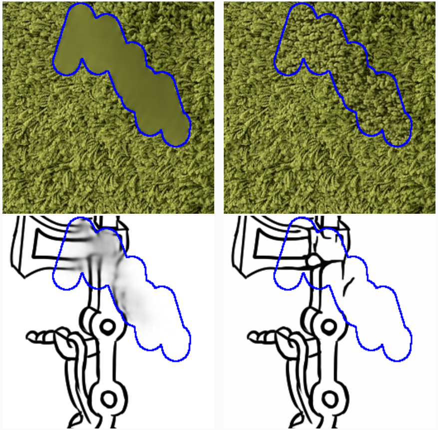
Diffusion inpainting



Results



Regression (left) vs Diffusion (right)



Regression (left) vs Diffusion (right)

Questions

References

DDPM
Ho, Jain, and Abbeel, *Denoising Diffusion Probabilistic Models*, 2020. Advances in Neural Information Processing Systems

CVPR 2022 tutorial on diffusion models:
`https://cvpr2022-tutorial-diffusion-models.github.io/`

Course on generative models by Valentin de Bortoli:
`https://vdeborto.github.io/project/generative_modeling/`

Denoising objective

When predicting x_0 or ϵ, we observe that the loss function is similar to a denoising problem using a classical objective or a residual objective:

$$\mathcal{L}_{x_0} = \sum_{t=1}^{T} w_{x_0}(t) \| x_0 - f_\theta(x_t, t) \|^2 \qquad \mathcal{L}_{\epsilon} = \sum_{t=1}^{T} w_{\epsilon}(t) \| \epsilon - f_\theta(x_t, t) \|^2$$

Where x_t is the noisy and rescaled version of x_0 at step t sampled from $q\left(\mathrm{x}_t \mid \mathrm{x}_0\right)$

Link with score-based methods

In the case of denoising diffusion models, we minimize the following loss:

$$\mathcal{L} = \sum_{t=1}^{T} \frac{1}{\sigma_t^2} \|x - f(\sqrt{\alpha_t} x + \sqrt{1-\alpha_t}\epsilon, t)\|^2$$

Which looks like the denoising loss for different variances σ_t^2 and a denoising network g:

$$\mathcal{L}_{\text{denoising}} = \sum_{t=1}^{T} \frac{1}{2\sigma_t^2} \|x - g(x + \sigma_t \epsilon, \sigma_t^2)\|^2$$

Link with score-based methods

The optimal denoiser for this denoising loss satisfies Tweedie's formula:

$$g_{\theta^*} = \arg\min_{\theta} \mathcal{L}_{\text{denoising}} \implies \boxed{\nabla \log p_{\sigma_t} = \frac{x - g_{\theta^*}(x + \sigma_t \epsilon)}{\sigma_t^2}}$$

We have access to the score, which is the gradient of the log-likelihood: $\nabla \log p_{\sigma_t}$

This gradient can be used during a Langevin process to sample from a distribution using only the gradients, starting from a point $x_0 \sim \mathcal{N}(0, I)$:

$$x_{i+1} = x_i + \gamma_i \nabla \log p_{\sigma_i}(x_i) + \sqrt{2\gamma_i} z_i$$

"gradient ascent with noise"