

GAN và diffusion

☑ Trắc nghiệm 27 📧 Thực hành 2 🔗 Chạy trên Kaggle ⬇ PDF bài

Model sinh ảnh (generative model) học từ một tập ảnh rồi tạo ra ảnh mới giống tập đó. Bài này học hai cách tiêu biểu. GAN cho hai mạng thi nhau: một mạng sinh ảnh giả, một mạng phân biệt ảnh thật với ảnh giả. Diffusion học cách khử nhiễu dần, bắt đầu từ một ảnh toàn nhiễu ngẫu nhiên. Cuối bài là cách đánh giá model sinh, vì vài ảnh trông đẹp chưa chứng minh được model sinh tốt.

MỤC TIÊU

- Hiểu vai trò của generator và discriminator trong GAN.
- Viết một bước huấn luyện GAN.
- Tính bước thêm nhiễu và mục tiêu học của diffusion.
- Sinh ảnh có điều kiện; so các cấu hình với seed và prompt cố định.

Sinh mẫu khác với dự đoán nhãn

Classifier ước lượng nhãn theo ảnh. Model sinh cần mô tả sự biến thiên của dữ liệu: những kiểu ảnh nào có thể xuất hiện và xuất hiện với tần suất nào. Nếu luôn trả cùng một ảnh đẹp, model bỏ mất sự đa dạng.

Nói theo xác suất, model phân biệt (discriminative) học $p(y | x)$, còn model sinh (generative) học chính phân phối dữ liệu $p(x)$, hoặc $p(x | c)$ khi có điều kiện c như một câu mô tả, để lấy được mẫu mới. Gõ “con mèo đội mũ len, vẽ màu nước” rồi nhận một bức ảnh mới, như DALL-E hay Stable Diffusion, là sinh ảnh từ văn bản.

Nguồn noise z không phải lỗi cần xóa: nó cung cấp biến thiên để cùng một prompt cho nhiều mẫu. Seed cố định dãy số ngẫu nhiên, nên hai seed cho hai noise khác nhau, còn cùng seed trong cùng điều kiện cho lại cùng thí nghiệm. “Cùng điều kiện” gồm cả model, sampler, phiên bản thư viện và phần cứng: đổi một thứ, ảnh có thể khác dù seed giữ nguyên.

HỎI NHANH

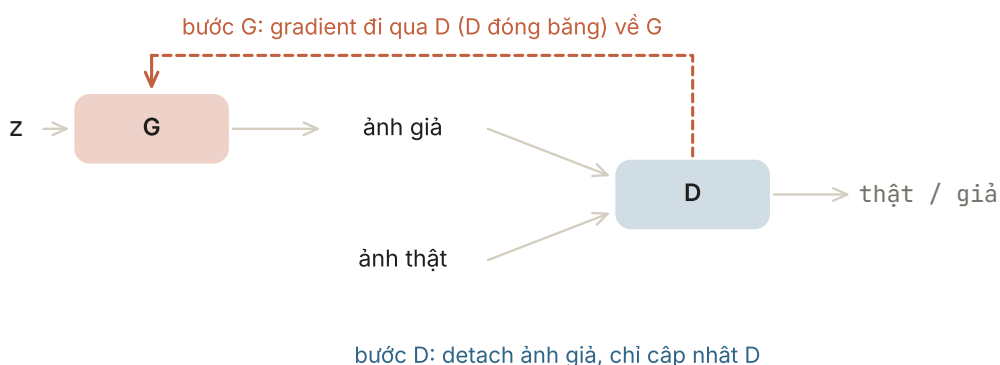
Một đội giữ nguyên seed 7, prompt và model, chỉ đổi sampler, chạy trên cùng một máy. Ảnh ra khác hẳn. Vì sao?

- A Seed chỉ cố định các số ngẫu nhiên, còn sampler đổi cách đi từ noise tới ảnh
- B Đổi sampler làm seed bị bỏ qua, nên noise ban đầu được lấy lại ngẫu nhiên
- C Cùng seed thì phải cùng ảnh, nên ảnh khác nghĩa là máy đang tính sai
- D Mỗi sampler đi kèm trọng số riêng, nên thực chất model đã bị đổi

GAN: hai mạng có hai mục tiêu

Generator $G(z)$ biến noise thành mẫu giả. Discriminator $D(x)$ học phân biệt mẫu thật với mẫu do generator tạo. Generator được cập nhật để discriminator đánh giá mẫu giả như thật. Hai mạng thay đổi cùng quá trình, nên mục tiêu mà một mạng thấy cũng đang thay đổi.

Khi dùng, chỉ còn generator chạy: trang web sinh khuôn mặt giả chỉ cần G biến noise thành ảnh, còn D chỉ có ích lúc train.



D học phân biệt ảnh thật với ảnh giả; G học qua gradient đi xuyên D .

Một cách train thông dụng dùng binary cross-entropy. Discriminator muốn nhận 1 cho thật, 0 cho giả:

$$L_D = -\mathbb{E}_x \log D(x) - \mathbb{E}_z \log(1 - D(G(z))).$$

Với non-saturating generator loss, generator muốn fake được D chấm 1:

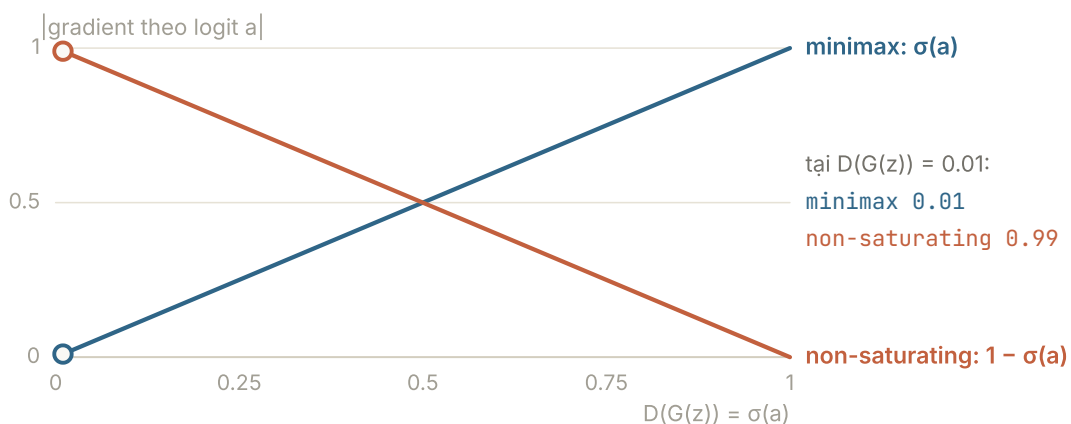
$$L_G = -\mathbb{E}_z \log D(G(z)).$$

Với $D(G(z)) = 0.2$, loss này là $-\log 0.2 \approx 1.609$; nếu G đẩy được điểm lên 0.8 thì loss còn khoảng 0.223. Phía D, chấm 0.9 cho ảnh thật và 0.2 cho ảnh giả cho loss $-\ln 0.9 - \ln 0.8 \approx 0.33$.

Cách viết minimax nguyên thủy cho G cực tiểu $\mathbb{E}_z \log(1 - D(G(z)))$. Hai cách cùng hướng đẩy $D(G(z))$ lên, nhưng khác nhau ở độ lớn gradient. Viết $D = \sigma(a)$ với a là logit của fake:

$$\frac{\partial}{\partial a} \log(1 - \sigma(a)) = -\sigma(a), \quad \frac{\partial}{\partial a} (-\log \sigma(a)) = -(1 - \sigma(a)).$$

Đầu train, D dễ nhận ra fake nên $D(G(z))$ gần 0. Khi đó cách viết minimax cho gradient gần 0 (bão hòa) và G gần như đứng yên, còn non-saturating cho gradient có độ lớn gần 1. Vì vậy code thực hành dùng $L_G = -\mathbb{E}_z \log D(G(z))$, tức BCE với nhãn “thật” cho fake.



Khi D bắt được ảnh giả, loss minimax gần như hết gradient; non-saturating thì không.

HỎI NHANH

Đầu train, D dễ nhận ra ảnh giả: $D(G(z)) = 0.01$. Tính theo logit a của ảnh giả, độ lớn gradient của $\log(1 - D)$, tức loss minimax, và của $-\log D$, tức loss non-saturating, lần lượt là bao nhiêu?

- A 0.99 và 0.01
- B 0.01 và 0.99
- C 0.01 và 4.61
- D Bằng nhau, vì hai loss cùng đẩy $D(G(z))$ lên

Một bước GAN thật, nhưng trên số một chiều

Trước khi train GAN ảnh, nên kiểm đồ thị gradient trên một bài nhỏ đến mức đọc được từng tham số: dữ liệu thật là bốn số quanh 2, G và D mỗi mạng là một lớp tuyến tính. Điều cần nhìn là tham số nào đổi ở bước nào; giá trị loss của một bước chưa nói gì về chất lượng sinh.

Python

```
import numpy as np
import torch
from torch import nn

torch.set_num_threads(1)
torch.manual_seed(58)
generator = nn.Linear(1, 1)
discriminator = nn.Linear(1, 1)
optimizer_g = torch.optim.SGD(generator.parameters(), lr=0.1)
optimizer_d = torch.optim.SGD(discriminator.parameters(), lr=0.1)
real = torch.tensor([[1.7], [1.9], [2.1], [2.3]])
noise = torch.tensor([[-1.], [0.], [1.], [2.]])
criterion = nn.BCEWithLogitsLoss()

g_before_d = generator.weight.detach().clone()
```

```

optimizer_d.zero_grad()
fake = generator(noise)
real_logits = discriminator(real)
fake_logits = discriminator(fake.detach())
loss_d = criterion(real_logits, torch.ones_like(real_logits))
loss_d += criterion(fake_logits, torch.zeros_like(fake_logits))
loss_d.backward()
optimizer_d.step()
print("G unchanged during D step:", torch.equal(g_before_d, generator.weight))
print("D loss:", round(loss_d.item(), 4))

```

D loss 1.4571 gần $2 \ln 2 \approx 1.386$, mức của một D chấm 0.5 cho mọi mẫu: D mới khởi tạo chưa phân biệt được thật và giả. `detach()` ở fake làm gradient của bước D không đi vào G. Bước tiếp theo giữ trọng số D nhưng vẫn cần đạo hàm của D theo đầu vào:

Python

```

d_before_g = discriminator.weight.detach().clone()
g_before_g = generator.weight.detach().clone()
for parameter in discriminator.parameters():
    parameter.requires_grad_(False)
optimizer_g.zero_grad()
fake = generator(noise)
fake_logits = discriminator(fake)
loss_g = criterion(fake_logits, torch.ones_like(fake_logits))
loss_g.backward()
optimizer_g.step()
for parameter in discriminator.parameters():
    parameter.requires_grad_(True)
print("D unchanged during G step:", torch.equal(d_before_g, discriminator.weight))
print("G changed during G step:", not torch.equal(g_before_g, generator.weight))
print("G loss:", round(loss_g.item(), 4))
print("hand loss for D(fake)=0.2:", round(-np.log(0.2), 4))

```

G loss 1.0656 ứng với trung bình nhân của $D(G(z))$ khoảng $e^{-1.0656} \approx 0.34$: D vẫn nghiêng về “giả”, nên G còn tín hiệu để học. Dòng cuối kiểm lại số tính tay ở mục trước.

Không đặt `no_grad()` quanh D trong bước G: như vậy sẽ cắt đường gradient cần về G. Đóng băng parameter D vẫn giữ phép tính theo fake. Ba dòng kiểm tham số phát hiện loại bug này trước khi chạy một GAN lớn.

GAN có thể dao động, discriminator quá mạnh hoặc generator chỉ sinh vài kiểu (mode collapse). Loss G/D không có quy tắc “càng thấp đồng thời càng tốt” giống supervised loss cố định. Cần xem chất lượng, đa dạng và metric phù hợp trên nhiều seed.

HỎI NHANH

Xóa `.detach()` ở bước D rồi chạy lại. Dòng kiểm vẫn in `G unchanged during D step: True`. Ta kết luận được gì?

- A Bỏ `detach()` là an toàn, vì phép kiểm đã in True
- B Trọng số G không đổi, nhưng `.grad` của G đã bị ghi
- C Phép kiểm sai, trọng số G thật ra đã bị cập nhật
- D Có `detach` hay không thì gradient bước D cũng không tới được G

HỎI NHANH

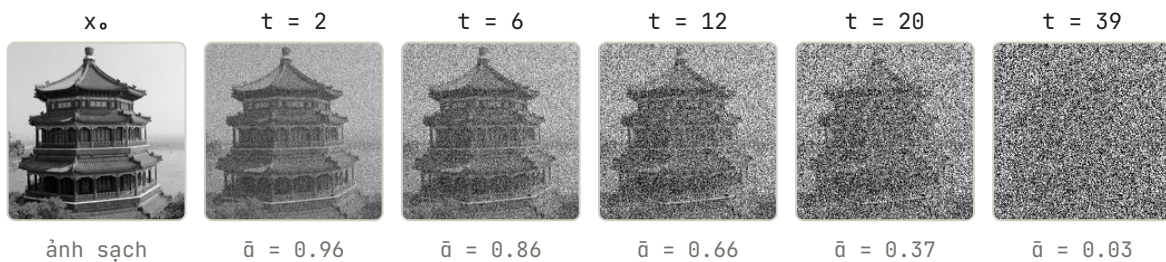
Ở bước G, ta bọc dòng `fake_logits = discriminator(fake)` trong `torch.no_grad()`, còn `fake = generator(noise)` vẫn ở ngoài. Chuyện gì xảy ra?

- A G vẫn học, vì fake được tạo ngoài khối `no_grad`
- B Code chạy được nhưng chỉ cập nhật D, vì D nằm trong khối
- C `loss_g.backward()` báo lỗi, vì loss không còn nối tới G
- D G học chậm hơn, vì gradient qua D bị thu nhỏ lại

Diffusion: học sửa ảnh ở nhiều mức nhiễu

Thay vì đối đầu hai mạng, diffusion tạo một nhiệm vụ supervised từ dữ liệu: thêm noise đã biết vào mẫu sạch, rồi học đoán noise hoặc đại lượng tương đương. Lúc sinh, bắt đầu

từ noise và làm nhiều bước khử nhiễu theo scheduler đã định nghĩa.



forward: thêm nhiễu theo lịch β của bài; sinh ảnh là học đi ngược chiều này

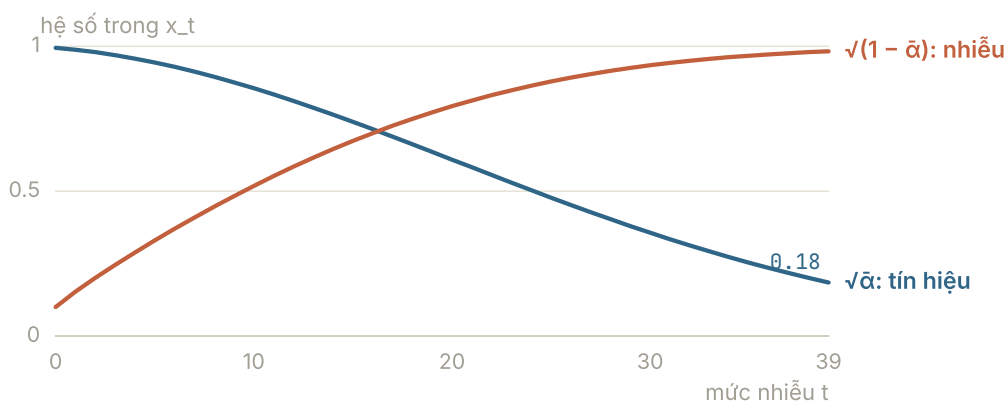
Thêm nhiễu dần qua 40 mức, đến cuối ảnh gần như chỉ còn nhiễu.

Đặt β_t là mức nhiễu mỗi bước, $\alpha_t = 1 - \beta_t$, $\bar{\alpha}_t = \prod_{s=1}^t \alpha_s$. Có thể lấy trực tiếp mẫu nhiễu ở thời điểm t :

$$x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon, \quad \epsilon \sim \mathcal{N}(0, I).$$

x_0 là dữ liệu sạch; t là mức noise, không phải epoch. Hệ số căn bậc hai phân phối phần phương sai cho tín hiệu và noise. Khi $\bar{\alpha}_t$ gần 1, giữ nhiều tín hiệu; gần 0, mẫu gần noise.

Noise predictor cần nhận cả x_t và t vì cùng một giá trị nhiễu ở mức khác có ý nghĩa khác; t thường được đưa vào mạng bằng embedding sin/cos như mã hóa vị trí ở bài 3.9. Khi $\bar{\alpha}_t \rightarrow 0$, x_t tiến về $\mathcal{N}(0, I)$ bất kể x_0 ; với $\beta = 0.02$ không đổi, sau 50 bước $\bar{\alpha} = 0.98^{50} \approx 0.36$.



Hệ số tín hiệu giảm dần, hệ số nhiễu tăng dần theo mức nhiễu.

Ví dụ: với $x_0 = 2$, $\bar{\alpha}_t = 0.64$, $\epsilon = 0.5$, ta có $x_t = 0.8 \cdot 2 + 0.6 \cdot 0.5 = 1.9$. Target khi train noise predictor là 0.5, không phải 2. Khi có dự đoán noise $\hat{\epsilon}$, có thể ước lượng sạch bằng $(x_t - \sqrt{1 - \bar{\alpha}_t}\hat{\epsilon})/\sqrt{\bar{\alpha}_t}$; đây chưa phải toàn bộ sampler.

Python

```
clean = 2.0
alpha_bar_example = 0.64
epsilon_example = 0.5
noisy = np.sqrt(alpha_bar_example) * clean
noisy += np.sqrt(1 - alpha_bar_example) * epsilon_example
estimated_clean = (noisy - np.sqrt(1 - alpha_bar_example) * epsilon_example)
estimated_clean /= np.sqrt(alpha_bar_example)
print("noisy:", round(noisy, 4))
print("reconstruction with known noise:", round(estimated_clean, 4))
```

Ta dùng noise thật ở dòng khôi phục, nên kết quả 2 là kiểm đại số, **không** phải model đã học khử nhiễu. Ở lúc sinh, ảnh sạch chưa biết và noise phải được model ước lượng.

HỎI NHANH

Trong ví dụ ở trên có $x_0 = 2$, $\bar{\alpha}_t = 0.64$, $\epsilon = 0.5$ và $x_t = 1.9$. Một mẫu train của noise predictor gồm đầu vào và target nào?

- A Đầu vào $(x_0 = 2, t)$, target $x_t = 1.9$
- B Đầu vào $x_t = 1.9$, bỏ t , target $\epsilon = 0.5$
- C Đầu vào $(x_t = 1.9, t)$, target $x_0 = 2$
- D Đầu vào $(x_t = 1.9, t)$, target $\epsilon = 0.5$

Train noise predictor trên CPU với dữ liệu một chiều

Thí nghiệm sau train thật một noise predictor trên dữ liệu một chiều: mỗi mẫu sạch nằm quanh -2 hoặc $+2$, mức nhiễu lấy ngẫu nhiên trong 40 mức (chỉ số 0–39), MLP nhận x_t

cùng chỉ số mức chia cho 39 và đoán ϵ . Cơ chế giống hệt khi train trên ảnh; chỉ khác là x_0 là một số thay cho tensor ảnh, và mạng là MLP thay cho U-Net.

Loss là MSE giữa ϵ và dự đoán. DDPM xuất phát từ mục tiêu likelihood; sau khi rút gọn, mỗi số hạng thành bình phương sai số giữa ϵ và $\epsilon_\theta(x_t, t)$ nhân một trọng số theo t , và bài báo bỏ trọng số ấy để được MSE đơn giản trên noise.

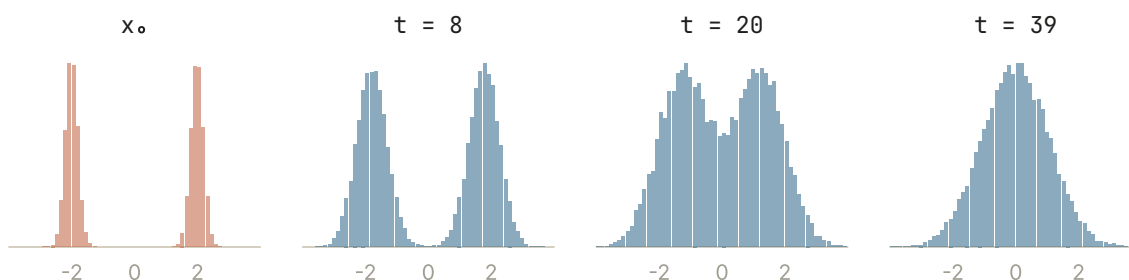
Python

```
torch.manual_seed(580)
steps = 40
betas = torch.linspace(0.01, 0.15, steps)
alphaBars = torch.cumprod(1 - betas, dim=0)
denoiser = nn.Sequential(nn.Linear(2, 24), nn.Tanh(), nn.Linear(24, 1))
optimizer = torch.optim.Adam(denoiser.parameters(), lr=0.01)

def make_noisy_batch(count):
    clean = (torch.randint(0, 2, (count, 1)) * 4 - 2).float()
    clean += 0.2 * torch.randn(count, 1)
    times = torch.randint(0, steps, (count,))
    eps = torch.randn(count, 1)
    kept = alphaBars[times, None]
    noisy = kept.sqrt() * clean + (1 - kept).sqrt() * eps
    inputs = torch.cat([noisy, times[:, None] / (steps - 1)], dim=1)
    return inputs, eps

valid_inputs, valid_noise = make_noisy_batch(128)
with torch.no_grad():
    before = nn.functional.mse_loss(denoiser(valid_inputs), valid_noise).item()
for update in range(250):
    inputs, targets = make_noisy_batch(64)
    optimizer.zero_grad()
    loss = nn.functional.mse_loss(denoiser(inputs), targets)
    loss.backward()
    optimizer.step()
with torch.no_grad():
    after = nn.functional.mse_loss(denoiser(valid_inputs), valid_noise).item()
    zero_baseline = valid_noise.square().mean().item()
print("noise MSE before/after:", round(before, 4), round(after, 4))
print("zero predictor MSE:", round(zero_baseline, 4))
print("last alpha_bar:", round(alphaBars[-1].item(), 4))
```

MSE giảm từ 0.9092 xuống 0.3434, thấp hơn nhiều so với baseline luôn đoán 0 (0.9815): model đã học. Ở mức cuối, $\bar{\alpha} = 0.0342$: tín hiệu chỉ còn hệ số $\sqrt{0.0342} \approx 0.18$, nên mẫu ở mức này đã gần noise chuẩn.



hai cụm quanh ± 2 nhòe dần thành một phân phối chuẩn

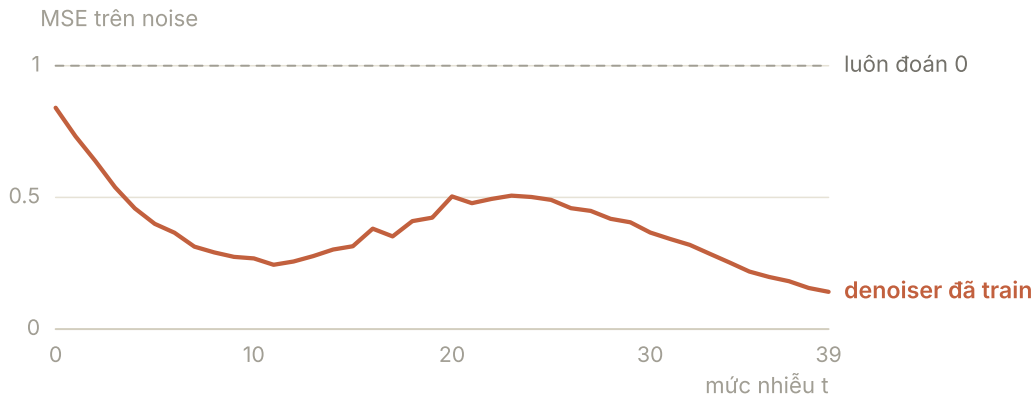
Hai cụm dữ liệu nhòe dần thành một phân phối chuẩn.

MSE không thể về 0, vì cùng một x_t có thể đến từ nhiều cặp (x_0, ϵ) . Ở mức nhiễu thấp, noise nhỏ hơn độ lệch tự nhiên của x_0 quanh ± 2 nên không tách được; ở mức giữa, không biết x_t đến từ cụm -2 hay $+2$. Con số này đo việc đoán noise; chất lượng mẫu sinh còn phụ thuộc sampler và cần đánh giá riêng.

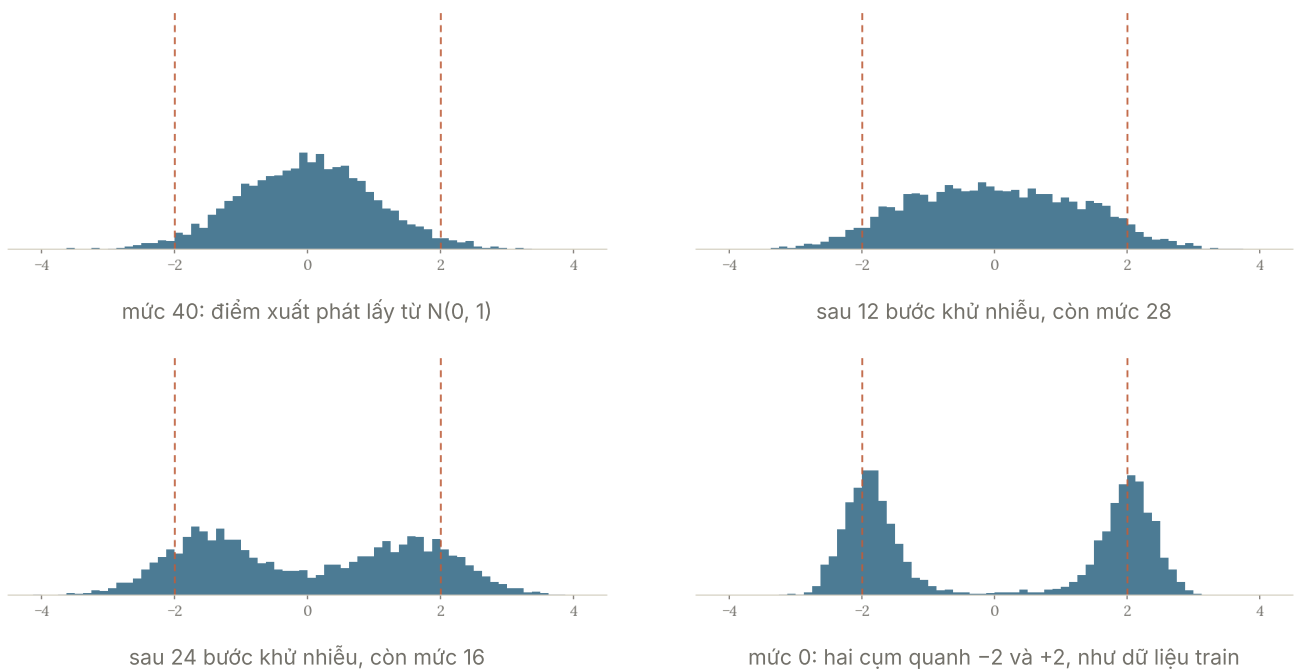
HỎI NHANH

Ở một mức t có $\bar{\alpha}_t = 0.001$, noise predictor đoán ϵ từ x_t dễ hay khó?

- A Dễ, vì $x_t \approx \epsilon$, gần như đọc thẳng được noise
- B Khó nhất, vì tín hiệu gần mất hết, không còn gì để dựa vào
- C Không xác định, vì phải biết x_0 mới tính được ϵ
- D Khó, vì noise lớn nên MSE ở mức này luôn lớn nhất



Sai số đoán nhiễu theo từng mức: khó ở mức thấp và mức giữa, dễ ở mức cao.



Sampler DDPM dùng noise predictor đã train của bài: 4000 điểm từ noise chuẩn dần tách về hai cụm ± 2 .

Từ noise predictor đến model sinh có điều kiện

Sampler bắt đầu từ $x_T \sim \mathcal{N}(0, I)$ và dùng dự đoán noise ở từng mức để đi về $x_{T-1}, \dots, x_0$; tùy scheduler, mỗi bước có thể thêm một lượng noise mới.

Lập công thức khôi phục x_0 ở trên không phải là DDPM: hệ số và phương sai của bước reverse phải khớp lịch nhiễu đã dùng khi train ([DDPM §3.2](#) viết đủ các hệ số này). Mỗi bước là một lần gọi denoiser, và bước sau cần kết quả bước trước: 50 bước là 50 lần forward nối tiếp cho một ảnh, trong khi generator GAN chỉ cần một.

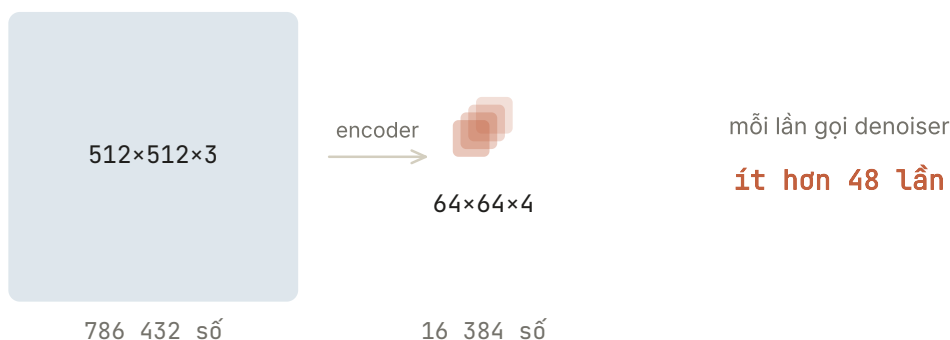
Diffusion có điều kiện chữ đưa embedding của prompt vào denoiser, thường qua cross-attention.

Latent diffusion (họ Stable Diffusion) chạy cả quá trình trong latent nén của ảnh: encoder của một autoencoder nén ảnh, khử nhiễu diễn ra trên latent, decoder biến latent cuối thành pixel. Stable Diffusion 1.x dùng latent $64 \times 64 \times 4$ cho ảnh $512 \times 512 \times 3$, nên mỗi lần gọi denoiser rẻ hơn nhiều.

HỎI NHANH

Dùng latent $64 \times 64 \times 4$ thay cho ảnh $512 \times 512 \times 3$. Mỗi lần gọi denoiser xử lý ít hơn bao nhiêu lần số giá trị?

- A 8 lần
- B 48 lần
- C 64 lần
- D 192 lần

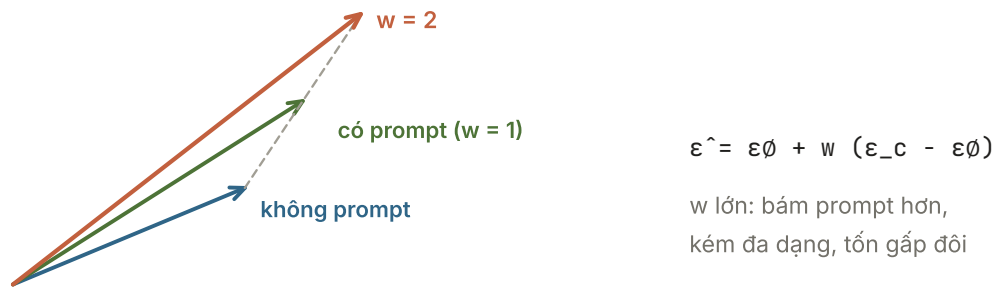


Khử nhiễu trên latent ít hơn 48 lần số so với ảnh gốc.

Classifier-free guidance (CFG) điều khiển mức bám prompt. Mỗi bước gọi denoiser hai lần, có prompt c và không prompt $\emptyset$, rồi đẩy dự đoán ra xa bản không điều kiện:

$$\hat{\epsilon} = \epsilon_{\theta}(x_t, t, \emptyset) + w [\epsilon_{\theta}(x_t, t, c) - \epsilon_{\theta}(x_t, t, \emptyset)].$$

$w = 1$ là dự đoán có điều kiện thường. w lớn hơn làm ảnh bám prompt hơn, đổi lại đa dạng giảm và màu, chi tiết có thể quá gắt; mỗi bước cũng tốn gấp đôi. Model học cả hai chế độ nhờ khi train bỏ prompt ở một phần mẫu. Ví dụ ϵ không prompt 0.1 và có prompt 0.4 với $w = 5$ cho $0.1 + 5 \cdot 0.3 = 1.6$. Số bước, sampler, kích thước, w , seed và prompt đều cần ghi khi so kết quả.



Guidance đi tiếp w lần theo hướng từ dự đoán không prompt sang có prompt.

Script của chương đưa quy trình này vào một lệnh: `local_assets.py --task diffusion` kiểm gói diffusers và transformers, thư mục pipeline có `model_index.json`, prompt và đường dẫn ảnh ra; không có `--run` thì chỉ kiểm điều kiện rồi dừng.

Với `--run`, script nạp pipeline (tokenizer, text encoder, U-Net, VAE, scheduler, weights safetensors) ở chế độ offline trên CPU float32 và sinh một ảnh 256×256 bằng 10 bước với seed 5, không ghi đè ảnh cũ. Một ảnh như vậy xác nhận đường nạp và inference chạy được; so chất lượng hai cấu hình cần cả bộ prompt và seed, như mục sau.

Đánh giá model sinh

FID so phân phối feature (lấy từ một mạng phân loại pretrained) của ảnh sinh với ảnh thật. FID càng thấp càng tốt, nhưng cần hàng nghìn mẫu và cách tính nhất quán giữa các lần so: 500 ảnh mỗi model là quá ít, và FID tính trên số mẫu khác nhau không so được với nhau.

Sinh dữ liệu để train classifier cần kiểm nhãn, trùng lặp và mức khớp miền ảnh thật. Model sinh học từ dữ liệu nguồn, có thể lặp nội dung hoặc thiên lệch; ảnh sinh không tự là bằng chứng về thế giới thật.

HỎI NHANH

Ta so hai cấu hình sampler. Cấu hình A cho 4 ảnh đẹp nhất chọn từ 200 lần chạy. Cấu hình B cho 4 ảnh của seed 0 đến 3. A trông đẹp hơn. So thế nào mới công bằng?

- A Chọn A, vì ảnh đẹp nhất cho thấy giới hạn trên của cấu hình
- B Lấy ảnh đẹp nhất trong 200 lần chạy của cả hai cấu hình rồi so
- C Dùng cùng prompt và seed, sinh nhiều mẫu, đo cả chất lượng lẫn đa dạng
- D Bỏ ảnh đi, so loss train của hai cấu hình

Khi đi thi

- Ước lượng thời gian: số ảnh $\times$ số bước $\times$ một lần gọi denoiser, gấp đôi khi bật CFG.
- So cấu hình trên cùng bộ prompt và seed, mỗi lần đổi một thứ.
- Ảnh sinh chỉ dùng cho train; validation và test giữ ảnh thật.

Tóm tắt

- Model sinh học phân phối dữ liệu; noise và seed tạo ra sự đa dạng.
- GAN: D học phân biệt thật/giả, G học lừa D; hai bước cập nhật luân phiên, phải đúng đường gradient (detach ở bước D) và kiểm đa dạng.
- Diffusion học đoán noise ở nhiều mức bằng MSE, rồi sinh bằng nhiều bước khử nhiễu theo scheduler.

- Latent diffusion làm mỗi bước rẻ hơn; guidance đổi đa dạng lấy mức bám prompt.
- Đánh giá model sinh cần nhiều mẫu, cùng prompt và seed; vài ảnh đẹp chưa đủ.

Kiểm tra cuối bài

CÂU 1

Forward diffusion hai bước: $x_1 = \sqrt{\alpha_1} x_0 + \sqrt{1 - \alpha_1} \epsilon_1$ và $x_2 = \sqrt{\alpha_2} x_1 + \sqrt{1 - \alpha_2} \epsilon_2$, với $\epsilon_1, \epsilon_2 \sim \mathcal{N}(0, 1)$ độc lập, $\alpha_1 = 0.9$, $\alpha_2 = 0.8$. Phân phối của x_2 khi biết x_0 có mean và phương sai bao nhiêu?

A Mean $\sqrt{0.72} x_0$, phương sai 0.28

B Mean $\sqrt{0.72} x_0$, phương sai 0.30

C Mean $0.72 x_0$, phương sai 0.28

D Mean $0.72 x_0$, phương sai 0.30

PHÒNG THEO [USA-NA-AIO 2026 · Vòng 2 · Ngày 2 · Part 3.1.1](#)

CÂU 2

Lịch nhiễu có $0 < \alpha_t \leq 0.99$ với mọi t . Khi $t \rightarrow \infty$, phân phối của x_t khi biết x_0 tiến tới đâu?

- A $\mathcal{N}(x_0, 1)$, tín hiệu giữ nguyên và noise cộng thêm vào
- B $\mathcal{N}(0, \infty)$, phương sai tăng không giới hạn
- C $\mathcal{N}(0, 1)$, không còn phụ thuộc vào x_0
- D $\mathcal{N}(0, 0)$, mọi mẫu dồn hết về điểm 0

PHÒNG THEO [USA-NA-AIO 2026 · Vòng 2 · Ngày 2 · Part 3.1.3](#)

CÂU 3

KL divergence đo độ khác nhau giữa hai phân phối, và bằng 0 khi chúng trùng nhau. Với hai Gaussian một chiều, $D_{KL}(\mathcal{N}(\mu_1, \sigma_1^2) \parallel \mathcal{N}(\mu_2, \sigma_2^2)) = \ln \frac{\sigma_2}{\sigma_1} + \frac{\sigma_1^2 + (\mu_1 - \mu_2)^2}{2\sigma_2^2} - \frac{1}{2}$. Ở một bước reverse, phân phối đích có mean 1.30, model có mean 1.10, hai bên cùng phương sai 0.04. KL bằng bao nhiêu?

- A 0.04
- B 0.5
- C 1.0
- D 2.5

PHÒNG THEO [USA-NA-AIO 2026 · Vòng 2 · Ngày 2 · Part 3.2.1–3.2.5](#)

CÂU 4

Noise predictor $\epsilon_{\theta}(x_t, t)$ cho ảnh là một U-Net. Đầu vào x_t có shape (B, C, H, W) , đầu ra cùng shape. Nên đưa bước t vào mạng bằng cách nào?

- A Bỏ t , vì mạng đủ sâu sẽ tự đọc mức nhiễu từ độ hạt của x_t
- B Train một mạng riêng cho mỗi t , lúc sinh thì chọn mạng theo t
- C Cộng t/T vào noise dự đoán ở đầu ra, sau khi mạng chạy xong
- D Mã hóa t thành vector rồi cộng vào feature map của các khối

PHÒNG THEO [USA-NA-AIO 2026 · Vòng 2 · Ngày 2 · Part 3.3](#)

CÂU 5

Đề cho 30 phút để sinh 600 ảnh trên một GPU. Generator của GAN tạo một ảnh trong 15 ms. Pipeline diffusion dùng 50 bước, mỗi bước gọi U-Net một lần mất 40 ms, không dùng CFG. Diffusion mất bao lâu để sinh xong, và vì sao chậm hơn GAN?

- A Khoảng 20 phút, vì mỗi ảnh cần 50 lần gọi mạng nối tiếp nhau
- B Khoảng 20 phút, vì model diffusion có nhiều tham số hơn
- C Khoảng 24 giây, vì mỗi ảnh chỉ cần một lần gọi U-Net
- D Khoảng 24 giây, vì các bước khử nhiễu chạy song song

PHÒNG THEO [NOAI 2026 Singapore · Vòng sơ loại · Câu 255](#)

CÂU 6

Ở một bước sinh, denoiser cho $\epsilon_\theta(x_t, t, \emptyset) = 0.2$ khi không có prompt, và $\epsilon_\theta(x_t, t, c) = 0.5$ khi có prompt. Với classifier-free guidance $w = 3$, noise dùng cho bước này là bao nhiêu? w lớn có tác dụng gì?

- A 1.5, và sampling nhanh hơn vì cần ít bước hơn
- B 1.5, và ảnh bám prompt hơn, đổi lại kém đa dạng
- C 1.1, và ảnh bám prompt hơn, đổi lại kém đa dạng
- D 1.1, và sampling nhanh hơn vì cần ít bước hơn

PHÒNG THEO NOAI 2026 Singapore · Vòng sơ loại · Câu 254

CÂU 7

GAN sinh khuôn mặt sắc nét, nhưng 64 mẫu từ 64 seed khác nhau gần như cùng một gương mặt, dù dữ liệu thật rất đa dạng. Điều này nói lên gì?

- A Ảnh sắc nét chứng tỏ G đã học được phân phối dữ liệu
- B Loss của G vẫn đang giảm, nên train thêm thì hiện tượng sẽ tự hết
- C Noise z có quá ít chiều, tăng số chiều của z chắc chắn sửa được
- D Đây là dấu hiệu mode collapse, cần đo độ đa dạng và độ phủ của phân phối

CÂU 8

Một đội viết `optimizer_g = torch.optim.SGD(list(generator.parameters()) + list(discriminator.parameters()), lr=0.1)`, và ở bước G không đóng băng D. Phần còn lại giữ như code của bài. Dòng kiểm nào lộ ra lỗi, và D bị đẩy theo hướng nào?

- A Không dòng nào lộ, vì `optimizer_g` vẫn chỉ cập nhật G
- B D unchanged during G step in False, và D bị dạy gọi ảnh giả là thật
- C D unchanged during G step in False, và D học tách ảnh giả rõ hơn
- D G changed during G step in False, vì gradient bị chia sang D

CÂU 9

Một noise predictor nhỏ có MSE đoán noise trên batch kiểm tra, cùng quy tắc sinh dữ liệu và cùng lịch nhiễu, giảm từ 1.0 xuống 0.6 sau khi train. Bằng chứng này cho phép kết luận điều gì?

- A Model đoán noise tốt hơn trên phân phối và lịch nhiễu đã kiểm
- B Ảnh do sampler dùng model này sinh ra đẹp hơn khoảng 40%
- C Model dùng được ngay với một lịch nhiễu khác mà không cần train lại
- D Sampler hoàn chỉnh đã chạy đúng, vì MSE noise đã giảm

CÂU 10

Đội có 300 ảnh thật có nhãn. Họ sinh thêm 3000 ảnh bằng diffusion theo prompt của từng lớp, trộn tất cả, rồi chia ngẫu nhiên 80/20 thành train và validation. Validation accuracy tăng từ 71% lên 90%. Con số này có đáng tin không?

- A Đáng tin, ảnh sinh giúp thêm 19 điểm, nên nộp ngay cấu hình này
- B Đáng tin, nên sinh thêm ảnh cho đến khi validation ngừng tăng
- C Đáng tin, vì validation có cả ảnh thật lẫn ảnh sinh
- D Chưa tin được, phải chấm lại trên validation chỉ gồm ảnh thật

Đọc thêm

- Goodfellow và cộng sự — [Generative Adversarial Networks](#): đọc hai vai trò và phân biệt minimax với loss generator thực hành.
- Ho, Jain, Abbeel — [Denoising Diffusion Probabilistic Models](#): đọc forward noise, noise prediction và reverse sampler.
- Rombach và cộng sự — [High-Resolution Image Synthesis with Latent Diffusion Models](#): đọc vì sao khử nhiễu trong latent và cách đưa prompt vào qua cross-attention.
- Ho, Salimans — [Classifier-Free Diffusion Guidance](#): đọc cách train một model cho cả hai chế độ và đánh đổi giữa mức bám điều kiện với đa dạng.
- [Diffusers — Stable Diffusion pipelines](#): đối chiếu điều kiện, sampler và cách nạp pipeline khi có asset.

[Tải gói recipe của chương](#) — gồm các module phụ và hướng dẫn chuẩn bị asset.