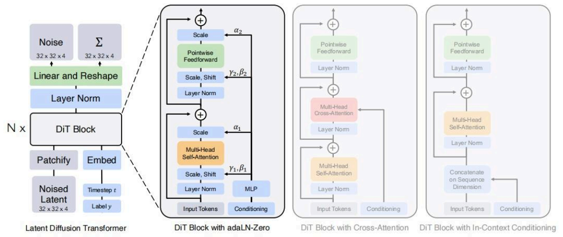
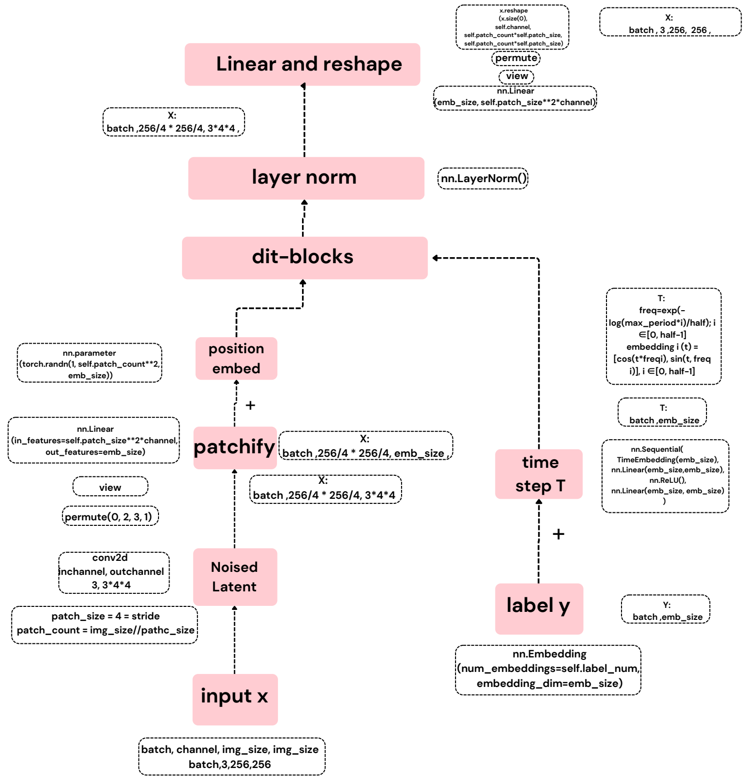


Figure 3. **The Diffusion Transformer (DiT) architecture.** *Left:* We train conditional latent DiT models. The input latent is decomposed into patches and processed by several DiT blocks. *Right:* Details of our DiT blocks. We experiment with variants of standard transformer blocks that incorporate conditioning via adaptive layer norm, cross-attention and extra input tokens. Adaptive layer norm works best.



x.reshape
(x.size(0),
self.channel,
self.patch_count*self.patch_size,
self.patch_count*self.patch_size)

permute

view

nn.Linear
(emb_size, self.patch_size**2*channel)

X:
batch , 3 ,256, 256 ,

Linear and reshape

X:
batch ,256/4 * 256/4, 3*4*4 ,

layer norm

nn.LayerNorm()

dit-blocks

nn.parameter
(torch.randn(1, self.patch_count**2, emb_size))

position embed

+

nn.Linear
(in_features=self.patch_size**2*channel, out_features=emb_size)

patchify

X:
batch ,256/4 * 256/4, emb_size ,

view

permute(0, 2, 3, 1)

X:
batch ,256/4 * 256/4, 3*4*4

conv2d
inchannel, outchannel
3, 3*4*4

Noised Latent

patch_size = 4 = stride
patch_count = img_size//pathc_size

input x

batch, channel, img_size, img_size
batch,3,256,256

T:
freq=exp(-log(max_period*i)/half); i ∈[0, half-1]
embedding i (t) = [cos(t*freqi), sin(t, freq i)], i ∈[0, half-1]

T:
batch ,emb_size

time step T

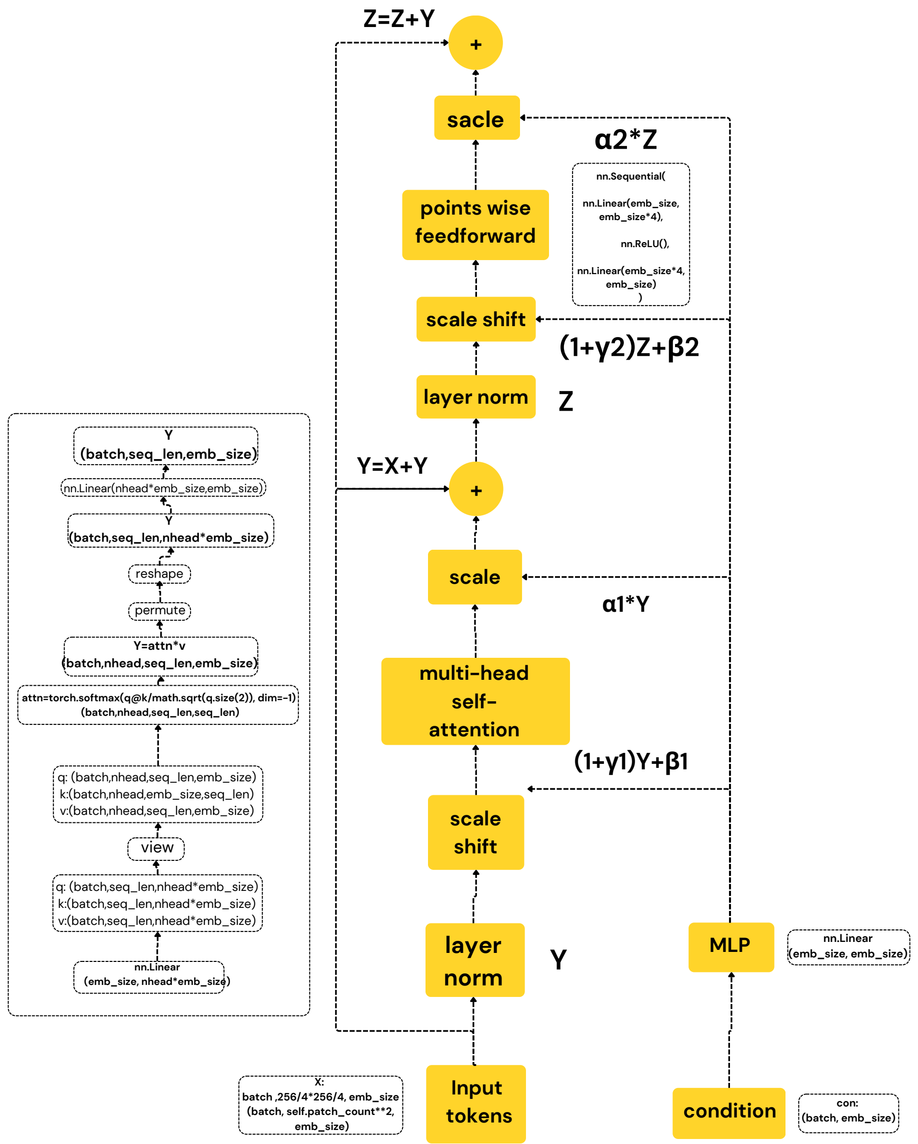
nn.Sequential(
TimeEmbedding(emb_size),
nn.Linear(emb_size,emb_size),
nn.ReLU(),
nn.Linear(emb_size, emb_size)
)

+

label y

Y:
batch ,emb_size

nn.Embedding
(num_embeddings=self.label_num, embedding_dim=emb_size)



Y
(batch,seq_len,emb_size)

nn.Linear(nhead*emb_size,emb_size)

Y
(batch,seq_len,nhead*emb_size)

reshape

permute

Y=attn*v
(batch,nhead,seq_len,emb_size)

attn=torch.softmax(q@k/math.sqrt(q.size(2)), dim=-1)
(batch,nhead,seq_len,seq_len)

q: (batch,nhead,seq_len,emb_size)
k:(batch,nhead,emb_size,seq_len)
v:(batch,nhead,seq_len,emb_size)

view

q: (batch,seq_len,nhead*emb_size)
k:(batch,seq_len,nhead*emb_size)
v:(batch,seq_len,nhead*emb_size)

nn.Linear
(emb_size, nhead*emb_size)

Z=Z+Y

+

sacle

α2*Z

nn.Sequential(
nn.Linear(emb_size, emb_size*4),
nn.ReLU(),
nn.Linear(emb_size*4, emb_size)
)

points wise feedforward

scale shift

(1+γ2)Z+β2

layer norm

Z

Y=X+Y

+

scale

α1*Y

multi-head self-attention

(1+γ1)Y+β1

scale shift

layer norm

Y

MLP

nn.Linear
(emb_size, emb_size)

X:
batch ,256/4*256/4, emb_size
(batch, self.patch_count**2, emb_size)

Input tokens

condition

con:
(batch, emb_size)

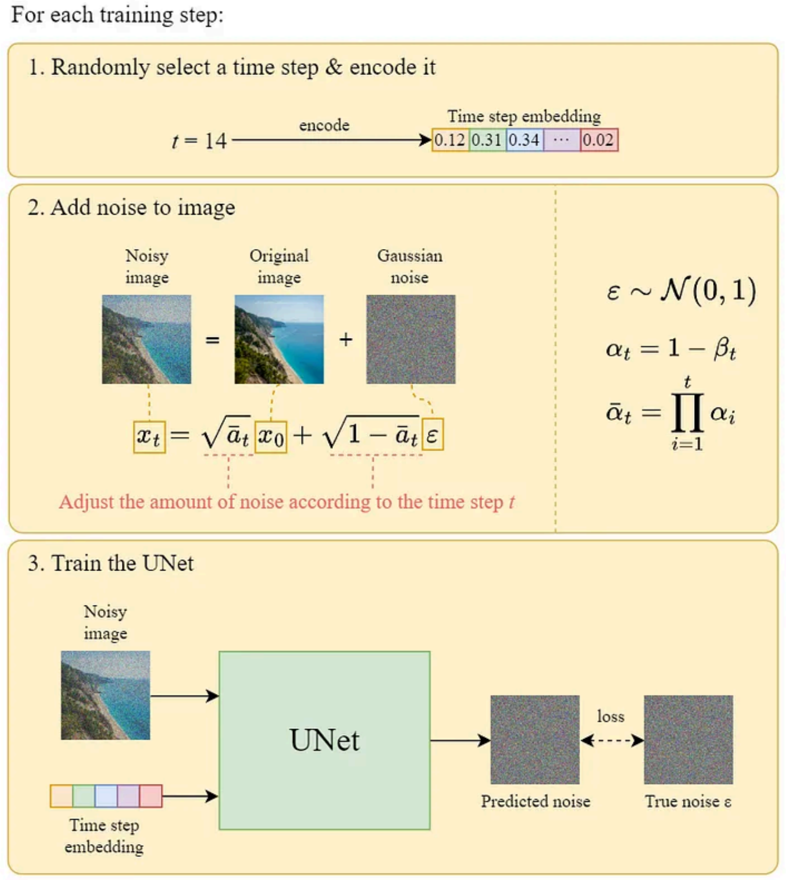
torch.cumprod(self.alpha, dim=-1)

self.alpha_cumprod[t].view(x.size(0), 1, 1, 1)

noise = torch.randn_like(x)

alphas_cumprod_prev=torch.cat((torch.tensor([1.0]),alphas_cumprod[:-1]),dim=-1) # alpha_t-1累乘 (T,),
[1,a1,a1*a2,a1*a2*a3,.....]

variance=(1-alphas)*(1-alphas_cumprod_prev)/(1-alphas_cumprod) # denoise用的方差 (T,)



For each training step:

1. Randomly select a time step & encode it

$t = 14 \xrightarrow{\text{encode}}$ Time step embedding 0.12 | 0.31 | 0.34 | ··· | 0.02

2. Add noise to image

Noisy image = Original image + Gaussian noise

$$x_t = \sqrt{\bar{a}_t}\, x_0 + \sqrt{1-\bar{a}_t}\, \varepsilon$$

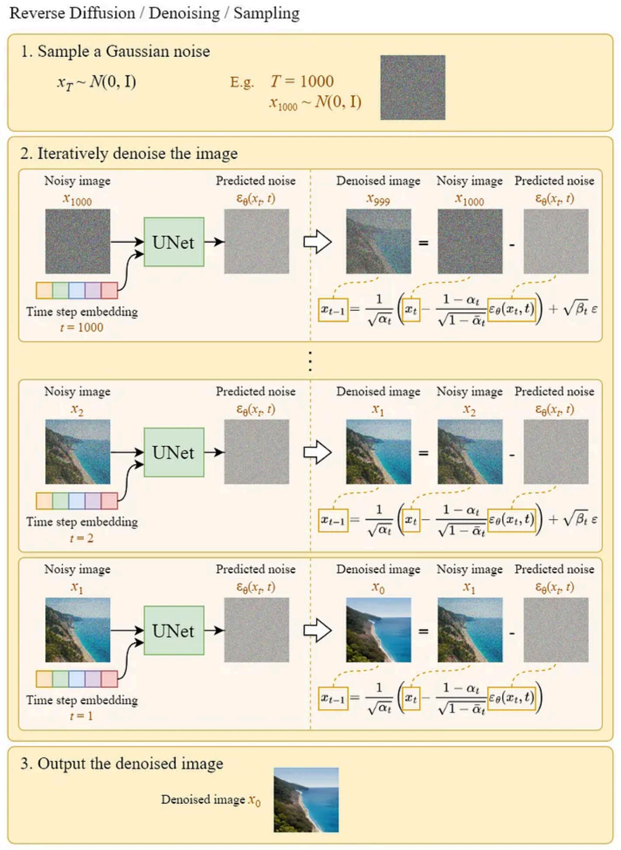
Adjust the amount of noise according to the time step t

$$\varepsilon \sim \mathcal{N}(0,1)$$

$$\alpha_t = 1 - \beta_t$$

$$\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$$

3. Train the UNet



Reverse Diffusion / Denoising / Sampling

1. Sample a Gaussian noise

$x_T \sim N(0, I)$ E.g. $T = 1000$
$x_{1000} \sim N(0, I)$

2. Iteratively denoise the image

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon_\theta(x_t,t)\right) + \sqrt{\beta_t}\,\varepsilon$$

⋮

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon_\theta(x_t,t)\right) + \sqrt{\beta_t}\,\varepsilon$$

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

3. Output the denoised image

Denoised image x_0