总逻辑

diffusion 过程包括 train 和 sample

——

train --> forward() --> self.p_losses() 随机生成的 noise 与 denoise_fn 输出的 x_recon 做 L1 L2 loss

——

sample --> diffusion.sample() --> p_sample_loop() --> p_sample() --> p_mean_variance() --> predict_start_from_noise(返回对 X0 的估计) | q_posterior(返回均值方差)

Train

——

```
def p_losses(self, x_in, noise=None):
    x_start = x_in['img_CLOUD'] # [6,3,128,128]
    x_start_ratio = x_in['img_CLOUD_ratio'] # [6,1,128,128]
    [b, c, h, w] = x_start.shape # [6,3,128,128]
    t = np.random.randint(1, self.num_timesteps + 1) # t : 1~2000
    continuous_sqrt_alpha_cumprod = torch.FloatTensor(
        np.random.uniform(
            self.sqrt_alphas_cumprod_prev[t-1],
            self.sqrt_alphas_cumprod_prev[t],
            size=b
        )
    ).to(x_start.device) # tensor continuous_sqrt_alpha_cumprod 1x6
    continuous_sqrt_alpha_cumprod = continuous_sqrt_alpha_cumprod.view(
        b, -1) # tensor continuous_sqrt_alpha_cumprod [6x1]

    noise = default(noise, lambda: torch.randn_like(x_start)) # tensor noise [6,3,128,128]
    x_noisy = self.q_sample( # tensor x_noisey [6,3,128,128] 返回是 Xt
        x_start=x_start, continuous_sqrt_alpha_cumprod=continuous_sqrt_alpha_cumprod.view(-1, 1, 1, 1), noise=noise)

    x_noisy_ratio = torch.cat([x_noisy, x_start_ratio], dim=1) # Xt + ratio [6,3,128,128] + [6,1,128,128] = [6,4,128,128]

    if not self.conditional:
```

```
        x_recon = self.denoise_fn(x_noisy, continuous_sqrt_alpha_cumprod)
    else: # 执行有 condtional = true
        x_recon = self.denoise_fn( # Unet 预测噪声 x_recon 问题：这里要不要加 x_start 加就是 7 通道 不加就是 4 通道
            x_noisy_ratio, continuous_sqrt_alpha_cumprod) # input[6,4,128,128] [6,1]
    loss = self.loss_func(noise, x_recon)
    return loss

def forward(self, x, *args, **kwargs): # x 是 train_data batch_size = 6
    return self.p_losses(x, *args, **kwargs)
```

代码逻辑：

P_losses()

1、取字典 x_in[‘img_CLOUD’]是 3 通道的输入图像；x_in['img_CLOUD_ratio']是云覆盖比例。

2、图像送入 q_sample 得到正向加噪声的 x_noisey namely ——full-noise X_t

3、x_noisy_ratio = torch.cat([x_noisy, x_start_ratio], dim=1) # [6,3,512,512] + [6,1,512,512] = [6,4,512,512]

4、送入——denoise_fn(Unet) 得到预测噪声 做 L1 | L2 loss

Sample

```python
def predict_start_from_noise(self, x_t, t, noise):
    return self.sqrt_recip_alphas_cumprod[t] * x_t - \
        self.sqrt_recipm1_alphas_cumprod[t] * noise

def q_posterior(self, x_start, x_t, t):
    posterior_mean = self.posterior_mean_coef1[t] * \
        x_start + self.posterior_mean_coef2[t] * x_t
    posterior_log_variance_clipped = self.posterior_log_variance_clipped[t]
    return posterior_mean, posterior_log_variance_clipped

def p_mean_variance(self, x, t, clip_denoised: bool, condition_x=None):
    batch_size = x.shape[0]
    noise_level = torch.FloatTensor(
        [self.sqrt_alphas_cumprod_prev[t+1]]).repeat(batch_size, 1).to(x.device)
    if condition_x is not None:
        # 根据 denoise_fn(Unet)预测出来的噪声对应公式中的 Zt 输入 predict_start_from_noise 函数 return X0 的估计量
        x_recon = self.predict_start_from_noise(
            x, t=t, noise=self.denoise_fn(torch.cat([condition_x, x], dim=1), noise_level)) # img + ratio
    else:
        x_recon = self.predict_start_from_noise(
            x, t=t, noise=self.denoise_fn(x, noise_level))

    if clip_denoised:
        x_recon.clamp_(-1., 1.)
    # DDPM 预测均值 方差固定（每 t 步）
    model_mean, posterior_log_variance = self.q_posterior( # q_posterior 根据 Xt、t、噪声ϵ 来估计原始图像 X0
        x_start=x_recon, x_t=x, t=t)
    return model_mean, posterior_log_variance

@torch.no_grad()
# 当前采样的得到的图像 return 每 t 步的图像
def p_sample(self, x, t, clip_denoised=True, condition_x=None):
    model_mean, model_log_variance = self.p_mean_variance(
        x=x, t=t, clip_denoised=clip_denoised, condition_x=condition_x)
    noise = torch.randn_like(x) if t > 0 else torch.zeros_like(x)
    return model_mean + noise * (0.5 * model_log_variance).exp()

@torch.no_grad()
def p_sample_loop(self, x_in, continous=True):
    device = self.betas.device # device(type='cuda', index=0)
    sample_inter = (1 | (self.num_timesteps//10)) # 2000//10 =201
```

```python
    if not self.conditional:
        shape = x_in
        img = torch.randn(shape, device=device)
        ret_img = img
        for i in tqdm(reversed(range(0, self.num_timesteps)), desc='sampling loop time step', total=self.num_timesteps):
            img = self.p_sample(img, i)
            if i % sample_inter == 0:
                ret_img = torch.cat([ret_img, img], dim=0)
    else:
        x = x_in['img_CLOUD']
        shape = x.shape
        img = torch.randn(shape, device=device) # 创造随机噪声
        ret_img = x # data_val['img_CLOUD']
        for i in tqdm(reversed(range(0, self.num_timesteps)), desc='sampling loop time step', total=self.num_timesteps):
            # 从 1999 到 0
            img = self.p_sample(img, i, condition_x=x_in['img_CLOUD_ratio']) #这个 codition_x 应该是 ratio
            if i % sample_inter == 0: # 打印 200 张加噪中间图
                ret_img = torch.cat([ret_img, img], dim=0)
    if continous:
        return ret_img
    else:
        return ret_img[-1]
```

P_sample()： 根据 mean,variance 还原 X_0

p_mean_variance()： 计算 mean,variance

1、predict_start_from_noise() 预测 X_0 的中间估计估计量 x_recon

2、

X：就是 XT 全噪声图像

t： 时间步骤 t

noise：denoise_fn(Unet)预测的 Zt 或者 ε(epsilon)

3、q_posterior() 根据 x_recon return 均值和方差

数学支撑：

`predict_start_from_noise` 函数

这个函数的目的是给定一个带噪声的图像 x_t 和噪声 ϵ，估计原始图像 x_0。在数学上，这可以表示为：

$$x_0 = \frac{1}{\sqrt{\alpha_t}} x_t - \frac{\sqrt{1-\alpha_t}}{\sqrt{\alpha_t}} \epsilon$$

其中 α_t 是时间步 t 下的累积乘积 $\prod_{s=1}^{t}(1-\beta_s)$，$\beta_s$ 是噪声调度中的参数。

self.sqrt_recip_alphas_cumprod[t]：

$$\sqrt{\frac{1}{\alpha_t}}$$

self.sqrt_recipm1_alphas_cumprod[t]：

$$\sqrt{\frac{1}{\alpha_t} - 1}$$

$$x_0 = \sqrt{\frac{1}{\alpha_t}} x_t - \sqrt{\frac{1}{\alpha_t} - 1} \cdot \epsilon$$

’q_posterior’函数

正向加噪声：

公式 1

$$x_t = \sqrt{\hat{\alpha}_t} x_0 + \sqrt{1 - \hat{\alpha}_t} \epsilon$$

公式 2

$$\hat{\mu}_t(x_t, x_0) := \frac{\sqrt{\hat{\alpha}_{t-1}}\beta_t}{1-\hat{\alpha}_t}x_0 + \sqrt{\frac{\alpha_t(1-\hat{\alpha}_{t-1})}{1-\hat{\alpha}_t}}x_t$$

公式 3

$$\begin{aligned}\hat{\mu}_t &= \frac{\sqrt{\alpha_t(1-\hat{\alpha}_{t-1})}}{1-\hat{\alpha}_t}x_t + \frac{\sqrt{\hat{\alpha}_{t-1}\beta_t}}{1-\hat{\alpha}_t}\frac{1}{\sqrt{\alpha_t}}(x_t - \sqrt{1-\hat{\alpha}_t}z_t) \\ &= \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\hat{\alpha}_t}}z_t\right)\end{aligned}$$

注： 根据公式 1、公式 2 合并计算得到公式 3