```
import math

import torch
from torch import nn
from torch.nn import functional as F


def make_beta_schedule(
    schedule, n_timestep, linear_start=1e-4, linear_end=2e-2, cosine_s=8e-3
):
    if schedule == "quad":
        betas = (
            torch.linspace(
                linear_start ** 0.5, linear_end ** 0.5, n_timestep, dtype=torch.float64
            )
            ** 2
        )

    elif schedule == "linear":
        betas = torch.linspace(
            linear_start, linear_end, n_timestep, dtype=torch.float64
        )

    elif schedule == "cosine":
        timesteps = (
            torch.arange(n_timestep + 1, dtype=torch.float64) / n_timestep + cosine_s
        )
        alphas = timesteps / (1 + cosine_s) * math.pi / 2
        alphas = torch.cos(alphas).pow(2)
        alphas = alphas / alphas[0]
        betas = 1 - alphas[1:] / alphas[:-1]
        betas = betas.clamp(max=0.999)

    return betas


def extract(input, t, shape):
    out = torch.gather(input, 0, t)
    reshape = [shape[0]] + [1] * (len(shape) - 1)
    out = out.reshape(*reshape)

    return out


def noise_like(shape, noise_fn, device, repeat=False):
    if repeat:
        resid = [1] * (len(shape) - 1)
        shape_one = (1, *shape[1:])

        return noise_fn(*shape_one, device=device).repeat(shape[0], *resid)

    else:
        return noise_fn(*shape, device=device)


class GaussianDiffusion(nn.Module):
    def __init__(self, betas):
        super().__init__()

        betas = betas.type(torch.float64)
        timesteps = betas.shape[0]
        self.num_timesteps = int(timesteps)

        alphas = 1 - betas
```

```
        alphas_cumprod = torch.cumprod(alphas, 0)
        alphas_cumprod_prev = torch.cat(
            (torch.tensor([1], dtype=torch.float64), alphas_cumprod[:-1]), 0
        )
        posterior_variance = betas * (1 - alphas_cumprod_prev) / (1 - alphas_cumprod)

        self.register("betas", betas)
        self.register("alphas_cumprod", alphas_cumprod)
        self.register("alphas_cumprod_prev", alphas_cumprod_prev)

        self.register("sqrt_alphas_cumprod", torch.sqrt(alphas_cumprod))
        self.register("sqrt_one_minus_alphas_cumprod", torch.sqrt(1 - alphas_cumprod))
        self.register("log_one_minus_alphas_cumprod", torch.log(1 - alphas_cumprod))
        self.register("sqrt_recip_alphas_cumprod", torch.rsqrt(alphas_cumprod))
        self.register("sqrt_recipm1_alphas_cumprod", torch.sqrt(1 / alphas_cumprod - 1))
        self.register("posterior_variance", posterior_variance)
        self.register(
            "posterior_log_variance_clipped",
            torch.log(posterior_variance.clamp(min=1e-20)),
        )
        self.register(
            "posterior_mean_coef1",
            (betas * torch.sqrt(alphas_cumprod_prev) / (1 - alphas_cumprod)),
        )
        self.register(
            "posterior_mean_coef2",
            ((1 - alphas_cumprod_prev) * torch.sqrt(alphas) / (1 - alphas_cumprod)),
        )

    def register(self, name, tensor):
        self.register_buffer(name, tensor.type(torch.float32))

    def q_sample(self, x_0, t, noise=None):
        if noise is None:
            noise = torch.randn_like(x_0)

        return (
            extract(self.sqrt_alphas_cumprod, t, x_0.shape) * x_0
            + extract(self.sqrt_one_minus_alphas_cumprod, t, x_0.shape) * noise
        )

    def p_loss(self, model, x_0, t, noise=None):
        if noise is None:
            noise = torch.randn_like(x_0)

        x_noise = self.q_sample(x_0, t, noise)
        x_recon = model(x_noise, t)

        return F.mse_loss(x_recon, noise)

    def predict_start_from_noise(self, x_t, t, noise):
        return (
            extract(self.sqrt_recip_alphas_cumprod, t, x_t.shape) * x_t
            - extract(self.sqrt_recipm1_alphas_cumprod, t, x_t.shape) * noise
        )

    def q_posterior(self, x_0, x_t, t):
        mean = (
            extract(self.posterior_mean_coef1, t, x_t.shape) * x_0
            + extract(self.posterior_mean_coef2, t, x_t.shape) * x_t
        )
        var = extract(self.posterior_variance, t, x_t.shape)
        log_var_clipped = extract(self.posterior_log_variance_clipped, t, x_t.shape)

        return mean, var, log_var_clipped
```

```
    def p_mean_variance(self, model, x, t, clip_denoised):
        x_recon = self.predict_start_from_noise(x, t, noise=model(x, t))

        if clip_denoised:
            x_recon = x_recon.clamp(min=-1, max=1)

        mean, var, log_var = self.q_posterior(x_recon, x, t)

        return mean, var, log_var

    def p_sample(self, model, x, t, noise_fn, clip_denoised=True, repeat_noise=False):
        mean, _, log_var = self.p_mean_variance(model, x, t, clip_denoised)
        noise = noise_like(x.shape, noise_fn, x.device, repeat_noise)
        shape = [x.shape[0]] + [1] * (x.ndim - 1)
        nonzero_mask = (1 - (t == 0).type(torch.float32)).view(*shape)

        return mean + nonzero_mask * torch.exp(0.5 * log_var) * noise

    @torch.no_grad()
    def p_sample_loop(self, model, shape, device, noise_fn=torch.randn):
        img = noise_fn(shape, device=device)

        for i in reversed(range(self.num_timesteps)):
            img = self.p_sample(
                model,
                img,
                torch.full((shape[0],), i, dtype=torch.int64).to(device),
                noise_fn=noise_fn,
            )

        return img
```