(1) Diffuser Design

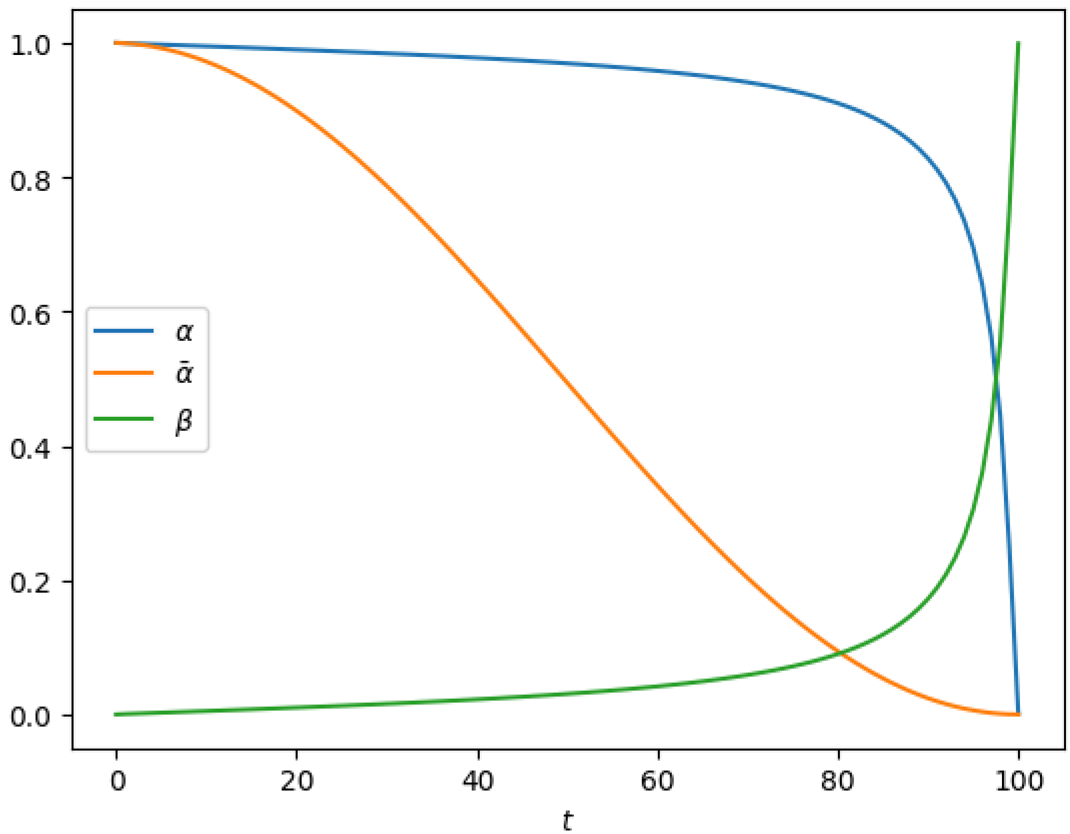
In [1]:

```
import torch
import torch.nn as nn
from tqdm.auto import tqdm
from typing import Union, Optional, Callable


class Diffuser:

    def __init__(
        self,
        diffusion_steps: int = 100,
        device: Optional[Union[str, torch.device]] = None,
    ):
        self.device = (
            torch.device("cuda" if torch.cuda.is_available() else "cpu")
            if device is None
            else device
        )
        self.steps = diffusion_steps
        s = 0.008
        tlist = torch.arange(1, diffusion_steps + 1, 1)
        temp1 = torch.cos((tlist / diffusion_steps + s) / (1 + s) * torch.pi / 2)
        temp1 = temp1 * temp1
        temp2 = torch.cos(((tlist - 1) / diffusion_steps + s) / (1 + s) * torch.pi / 2)
        temp2 = temp2 * temp2
        self.betas = 1 - (temp1 / temp2)
        self.betas[self.betas > 0.999] = 0.999
        self.betas = (
            torch.cat((torch.tensor([0]), self.betas), dim=0)
            .view([self.steps + 1, 1])
            .to(self.device)
        )  # set the first beta to 0
        self.generate_schedule()

    def generate_schedule(self):
        self.alphas = 1 - self.betas
        self.alphas_bar = torch.cumprod(self.alphas, 0)
```

```python
        self.one_minus_alphas_bar = 1 - self.alphas_bar
        self.sqrt_alphas = torch.sqrt(self.alphas)
        self.sqrt_alphas_bar = torch.sqrt(self.alphas_bar)
        self.sqrt_one_minus_alphas_bar = torch.sqrt(self.one_minus_alphas_bar)

    def forward_diffusion(
        self, x_0: torch.Tensor, t: Union[torch.Tensor, int], noise: torch.Tensor
    ) -> torch.Tensor:
        return self.sqrt_alphas_bar[t] * x_0 + self.sqrt_one_minus_alphas_bar[t] * noise

    def reverse_diffusion(
        self, x_t: torch.Tensor, t: Union[torch.Tensor, int], noise: torch.Tensor
    ) -> torch.Tensor:
        with torch.no_grad():
            coef1 = 1 / self.sqrt_alphas[t]
            coef2 = self.betas[t] / self.sqrt_one_minus_alphas_bar[t]
            sig = (
                torch.sqrt(self.betas[t])
                * self.sqrt_one_minus_alphas_bar[t - 1]
                / self.sqrt_one_minus_alphas_bar[t]
            )
            x_t = coef1 * (x_t - coef2 * noise) + sig * torch.randn_like(x_t)
            return torch.clamp(x_t, -3, 3)
```

In [10]:

```python
import matplotlib.pyplot as plt

diffuser = Diffuser(diffusion_steps=100)
plt.plot(diffuser.alphas[:, 0].cpu(), label=r"$\alpha$")
plt.plot(diffuser.alphas_bar[:, 0].cpu(), label=r"$\bar{\alpha}$")
plt.plot(diffuser.betas[:, 0].cpu(), label=r"$\beta$")
plt.legend()
plt.xlabel("$t$")
plt.show()
```

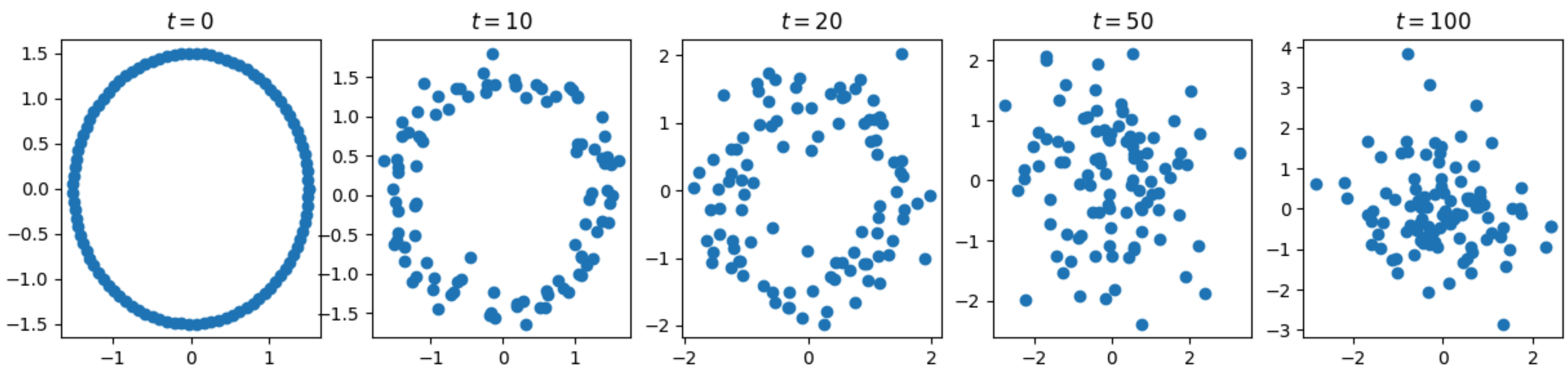


When t is close to T, $1/\sqrt{\alpha_t} \approx +\infty$, $\frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}} \approx 1$. Thus the prediction of $\mu_\theta\left(\mathbf{x}_t, t\right)$ turns to an infinitely magnified prediction errors, i.e., $\infty\left(\epsilon_T - \epsilon_\theta\left(\mathbf{x}_T, T\right)\right)$. This might make the predicted previous step $\mathbf{x}_{t-1}$ far from the true previous step $\mathbf{x}_{t-1}$.

(2) Circle Dataset

In [3]:
```
class Circle:

    def __init__(self, n_points: int = 100, radius: float = 1.5):
        self.theta = torch.linspace(0, 2 * torch.pi, n_points)
        self.x = radius * torch.cos(self.theta)
        self.y = radius * torch.sin(self.theta)
        self.points = torch.stack((self.x, self.y), dim=1)
```

```
    def __len__(self):
        return self.points.shape[0]

    def __getitem__(self, index):
        return self.points[index]


points = Circle().points.to(diffuser.device)
fig, axes = plt.subplots(1, 5, figsize=(15, 3))
for ax, t in zip(axes, [0, 10, 20, 50, 100]):
    diffused_points = diffuser.forward_diffusion(points, t, torch.randn_like(points))
    ax.scatter(
        diffused_points[:, 0].cpu(),
        diffused_points[:, 1].cpu(),
    )
    ax.set_title(f"$t={t}$")
plt.show()
```



(3) Diffusion Training

In [ ]:

```
import torch.nn as nn
from typing import List


class Net(nn.Module):
    def __init__(
        self,
```

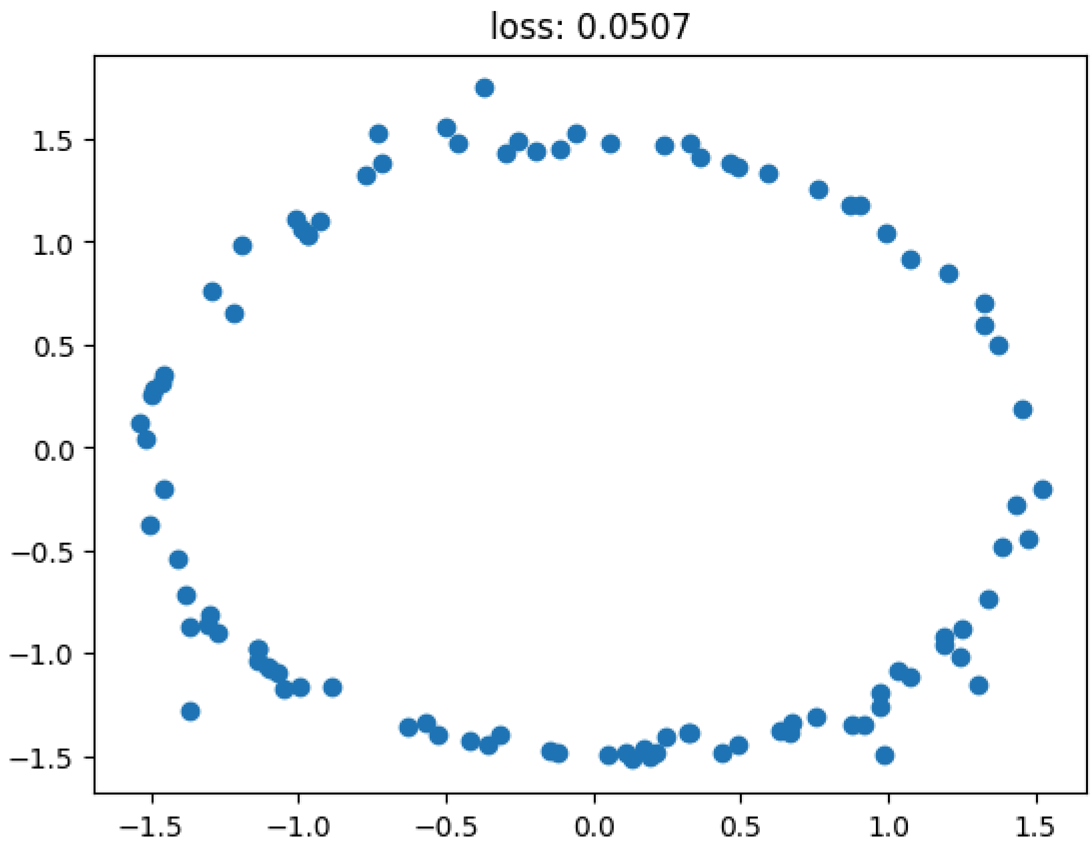
```python
        in_dim: int = 2,
        out_dim: int = 2,
        h_dims: List[int] = [512] * 4,
        dim_time=32,
    ):
        super().__init__()
        self.t_embeddings_base = (0.00001) ** (torch.linspace(0, 1, dim_time // 2))
        self.in_layer = nn.Linear(in_dim, h_dims[0] - dim_time)
        ins = h_dims
        outs = h_dims[1:] + [out_dim]
        self.layers = nn.ModuleList(
            [
                nn.Sequential(nn.LeakyReLU(), nn.Linear(in_d, out_d))
                for in_d, out_d in zip(ins, outs)
            ]
        )

    def time_encoder(self, t: torch.Tensor) -> torch.Tensor:
        embeddings = t[:, None] * self.t_embeddings_base.to(t.device)[None, :]
        return torch.cat((embeddings.sin(), embeddings.cos()), dim=-1)

    def forward(self, x: torch.Tensor, t: torch.Tensor) -> torch.Tensor:
        x = torch.cat([self.in_layer(x), self.time_encoder(t)], dim=-1)
        for layer in self.layers:
            x = layer(x)
        return x
```

In [ ]:

```python
n_epoch = 1000
diffuser = Diffuser(diffusion_steps=100)
net = Net().to(diffuser.device)
data_loader = torch.utils.data.DataLoader(
    Circle(100), batch_size=25, shuffle=True, drop_last=True
)
optimizer = torch.optim.Adam(net.parameters(), lr=1e-3)
p_bar = tqdm(range(n_epoch))
for epoch in p_bar:
    net.train()
    for x_0 in data_loader:
        optimizer.zero_grad()
        x_0 = x_0.to(diffuser.device)
        t = torch.randint(
            1, diffuser.steps + 1, size=(x_0.shape[0],), device=diffuser.device
```

```python
        )
        noise = torch.randn_like(x_0)
        x_t = diffuser.forward_diffusion(x_0, t, noise)
        predicted_noise = net(x_t, t)
        loss = ((predicted_noise - noise) ** 2).mean()
        loss.backward()
        optimizer.step()
        p_bar.set_description(f"loss:{loss.item():.4f}")
```

In [8]:
```python
def sample(
    diffuser, network: Union[nn.Module, Callable], x_t: torch.Tensor
) -> torch.Tensor:
    t = torch.tensor([diffuser.steps], device=diffuser.device).repeat(x_t.shape[0])
    for step in range(diffuser.steps):
        noise = network(x_t, t)
        x_t = diffuser.reverse_diffusion(x_t, t, noise)
        t = t - 1
    return x_t


circle_loss = lambda x_0: torch.abs(
    (torch.sqrt(x_0[:, 0] ** 2 + x_0[:, 1] ** 2) - 1.5)
).mean()

sampled = sample(diffuser, net, torch.randn(100, 2).to(diffuser.device))
plt.scatter(
    sampled[:, 0].cpu(),
    sampled[:, 1].cpu(),
)
plt.title(f"loss: {circle_loss(sampled):.4f}")
plt.show()
```

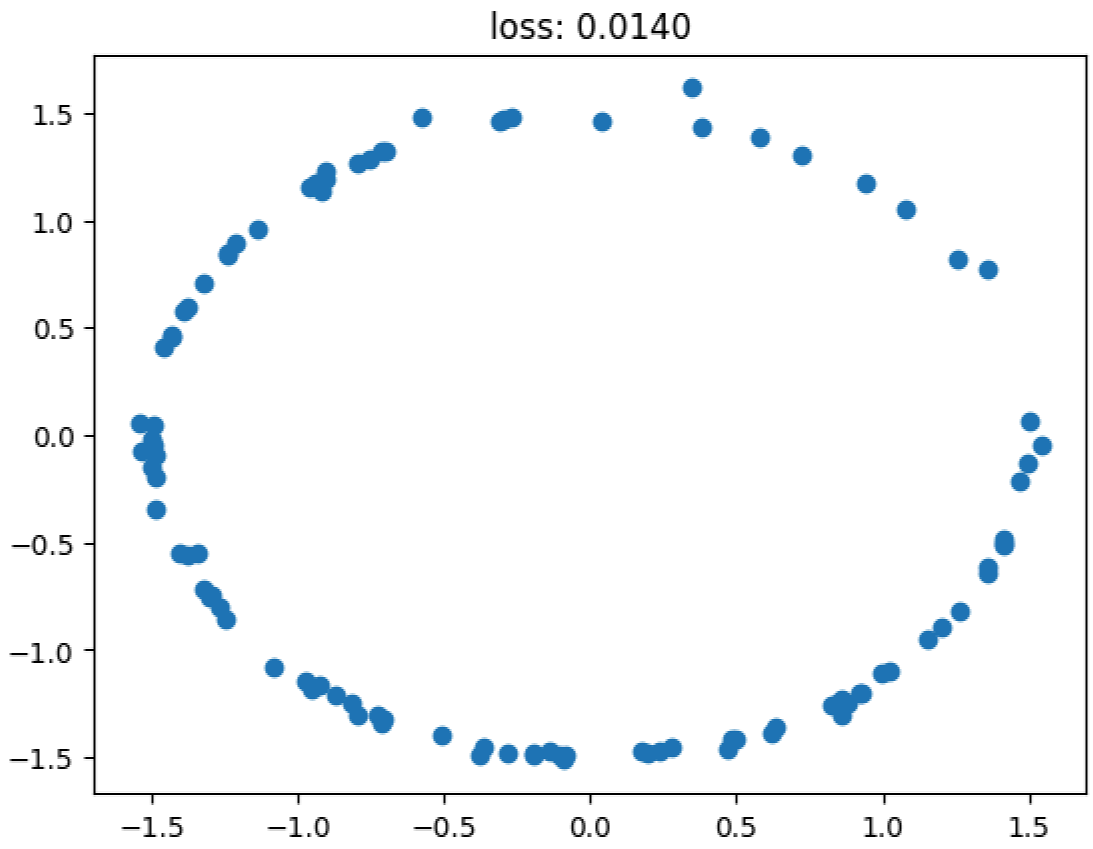


(4) Physics-based Diffusion Sampling

In [9]:

```
def phy_sample(
    diffuser,
    network: Union[nn.Module, Callable],
    x_t: torch.Tensor,
) -> torch.Tensor:
    t = torch.tensor([diffuser.steps], device=diffuser.device).repeat(x_t.shape[0])
    for step in range(diffuser.steps):
        noise = network(
            x_t,
            t,
        )
        x_t = diffuser.reverse_diffusion(x_t, t, noise)
```

```
        t = t - 1
        x_t.requires_grad = True
        x_0 = x_t - diffuser.sqrt_one_minus_alphas_bar[t] * noise.detach()
        x_0 = x_0 / diffuser.sqrt_alphas_bar[t]
        loss = circle_loss(x_0)
        loss.backward()
        x_t = (x_t - x_t.grad).detach()
    return x_t


sampled = phy_sample(diffuser, net, torch.randn(100, 2).to(diffuser.device))
plt.scatter(
    sampled[:, 0].cpu(),
    sampled[:, 1].cpu(),
)
plt.title(f"loss: {circle_loss(sampled):.4f}")
plt.show()
```


loss: 0.0140
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5