LAB 1-DDPM

314552036 周子翔

Task1

I. Complete and Explain of all #TODO code

1. Todo1

```
######## TODO ########
# DO  (variable) layers: Any  de this part.        Sean, 22 hours ago • i
self.layers = nn.ModuleList()
prev_dim = dim_in
for hid_dim in dim_hids:
    self.layers.append(TimeLinear(prev_dim, hid_dim, num_timesteps))
    prev_dim = hid_dim
self.out_layer = nn.Linear(prev_dim, dim_out)
######################
```

- It stacks several TimeLinear layers, which are linear layers modulated by a time embedding (so the network can use timestep information).
- Each hidden layer's size is specified by dim_hids.
- The final output layer maps the last hidden features to the output dimension (usually the same as the input, for noise prediction).

```
######## TODO ########
# DO NOT change the code outside this part.
h = x
for layer in self.layers:
    h = layer(h, t)
    h = F.silu(h)
out = self.out_layer(h)
######################
```

- The input x (noisy data) is passed through each layer, with the current timestep t provided to each TimeLinear layer.
- After each layer, a SiLU activation is applied for non-linearity.
- The final output is produced by the last linear layer, which predicts the noise for the given input and timestep.

2. Todo2

```
######## TODO ########
# DO NOT change the code outside this part.
# Compute xt.
alphas_prod_t = extract(self.var_scheduler.alphas_cumprod, t, x0)
# DDPM公式: x_t = sqrt(alpha_bar_t) * x0 + sqrt(1 - alpha_bar_t) * noise
xt = torch.sqrt(alphas_prod_t) * x0 + torch.sqrt(1 - alphas_prod_t) * noise

#######################
```

- For a given clean image x0 and timestep t, it adds noise according to the DDPM formula.
- alphas_prod_t is the cumulative product of alphas up to timestep t.
- The output xt is a noisy version of x0 at timestep t.

3. **Todo 3**

```
######## TODO ########
# DO NOT change the code outside this part.
# compute x_t_prev.
if isinstance(t, int):
    t = torch.tensor([t]).to(self.device)

# 1. predict noise
eps_pred = self.network(xt, t)

# 2. Posterior mean
beta_t      = extract(self.var_scheduler.betas,          t, xt)         # β_t
alpha_t     = extract(self.var_scheduler.alphas,         t, xt)         # α_t = 1 - β_t
alpha_bar_t = extract(self.var_scheduler.alphas_cumprod, t, xt)         # \bar{α}_t
t_prev      = (t - 1).clamp(min=0)
alpha_bar_t_prev = extract(self.var_scheduler.alphas_cumprod, t_prev, xt) # \bar{α}_{t-1}
mean = (
    torch.sqrt(alpha_bar_t_prev) * (xt - torch.sqrt(1 - alpha_bar_t) * eps_pred) / torch.sqrt(alpha_bar_t)
    + torch.sqrt(1 - alpha_bar_t_prev) * eps_pred
)

# 3. Posterior variance
var = beta_t * (1 - alpha_bar_t_prev) / (1 - alpha_bar_t)

# 4. Reverse step
if (t == 0).all():
    x_t_prev = mean
else:
    noise = torch.randn_like(xt)
    x_t_prev = mean + torch.sqrt(var) * noise

#######################
return x_t_prev
```

- Predicts the noise in xt using the network.
- Calculates the mean and variance for the posterior distribution at timestep t.
- Samples the previous timestep x_{t-1} using the mean and variance, adding noise unless at the final step.

4. **Todo 4**

```python
######## TODO ########
# DO NOT change the code outside this part.
# sample x0 based on Algorithm 2 of DDPM paper.
xt = torch.randn(shape).to(self.device)
num_timesteps = self.var_scheduler.num_train_timesteps
for i in reversed(range(num_timesteps)):
    t = torch.full((xt.shape[0],), i, dtype=torch.long, device=self.device)
    xt = self.p_sample(xt, t)
x0_pred = xt
######################
```

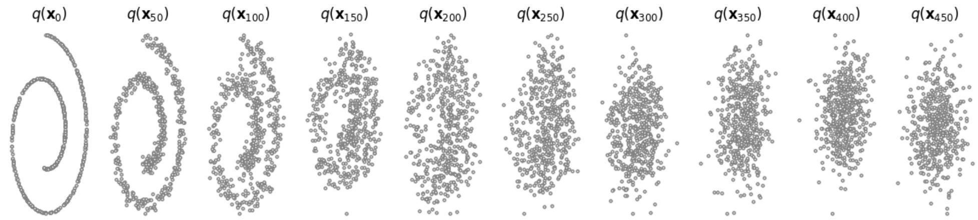
- Starts from pure noise.
- Iteratively denoises through all timesteps using p_sample.
- Returns the final denoised sample.

5. **Todo 5**

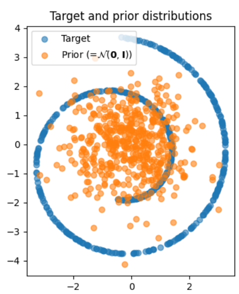
```python
######## TODO ########
# DO NOT change the code outside this part.
# compute noise matching loss.
batch_size = x0.shape[0]

# 1) random choose timestep
t = (
    torch.randint(0, self.var_scheduler.num_train_timesteps, size=(batch_size,))
    .to(x0.device)
    .long()
)

# 2) get GT noise, and use q_sample to get x_t
eps = torch.randn_like(x0)
xt = self.q_sample(x0, t, noise=eps)

# 3) predict noise
eps_pred = self.network(xt, t)

# 4) MSE loss (eps, eps_pred)
loss = F.mse_loss(eps_pred, eps)

######################
```

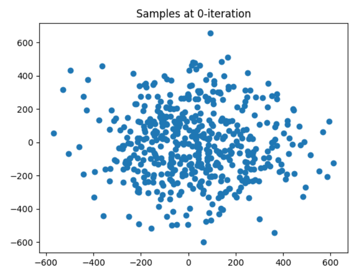
- Randomly selects a timestep for each sample in the batch.
- Adds noise to the clean data using q_sample.
- Predicts the noise using the network.
- Computes the mean squared error between predicted and true noise.

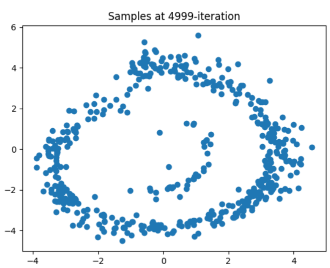
II. Fig of your implementation of q_sample (10pts)

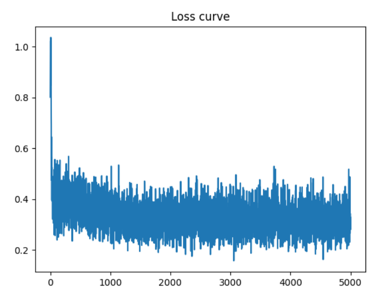


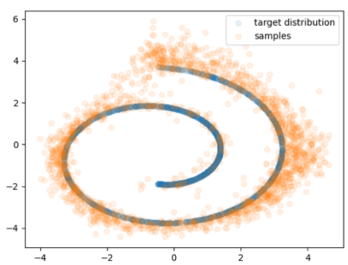
III. Fig of loss curve and evaluation result (10 pts)











DDPM Chamfer Distance: 440.0501

The DDPM implementation successfully learns the 2D swiss roll distribution, as shown by the sample plot and a reasonable Chamfer Distance.

Task 2

IV. Complete and Explain of all #TODO code.

1. Todo 1

```python
######## TODO ########
# Assignment 1. Implement the DDPM forward step.
alpha_bar_t = extract(self.alphas_cumprod, t, x_0)
x_t = torch.sqrt(alpha_bar_t) * x_0 + torch.sqrt(1 - alpha_bar_t) * eps
#######################
```

- Adds noise to the clean image according to the DDPM formula for a given timestep.

2. **Todo 4**

```python
######## TODO ########
# Implement the cosine beta schedule (Nichol & Dhariwal, 2021).
# Hint:
# 1. Define alphā_t = f(t/T) where f is a cosine schedule:
#       alphā_t = cos^2( ( (t/T + s) / (1+s) ) * (π/2) )
#    with s = 0.008 (a small constant for stability).
# 2. Convert alphā_t into betas using:
#       beta_t = 1 - alphā_t / alphā_{t-1}
# 3. Return betas as a tensor of shape [num_train_timesteps].
s = 0.008
steps = num_train_timesteps
t_arr = torch.arange(steps + 1, dtype=torch.float64)
f = lambda t: torch.cos(((t / steps + s) / (1 + s)) * np.pi / 2) ** 2
alphas_cumprod = f(t_arr)
betas = 1 - (alphas_cumprod[1:] / alphas_cumprod[:-1])
betas = torch.clip(betas, 0, 0.999)
betas = betas.to(torch.float32)
```

- It creates a smooth schedule for the cumulative product of alphas using a cosine function.
- Betas are then derived from the ratio of consecutive alphā values.
- This schedule can improve sample quality compared to linear/quadratic schedules.

3. **Todo 5**

```python
if predictor == "noise": #### TODO
    return self.step_predict_noise(x_t, t, net_out)
elif predictor == "x0": #### TODO
    return self.step_predict_x0(x_t, t, net_out)
elif predictor == "mean": #### TODO
    return self.step_predict_mean(x_t, t, net_out)
```

```python
######## TODO ########
# 1. Extract beta_t, alpha_t, and alpha_bar_t from the scheduler.
if isinstance(t, int):
    t = torch.tensor([t], device=x_t.device)
beta_t = extract(self.betas, t, x_t)
alpha_t = extract(self.alphas, t, x_t)
alpha_bar_t = extract(self.alphas_cumprod, t, x_t)
alpha_bar_t_prev = extract(torch.cat([self.alphas_cumprod.new_ones(1), self.alphas_cumprod[:-1]], 0), t, x_t)
# 2. Compute the predicted mean μ_θ(x_t, t) = 1/√α_t * (x_t - (β_t/√(1-ᾱ_t)) * ε̂_θ).
mean = (1 / torch.sqrt(alpha_t)) * (x_t - (beta_t / torch.sqrt(1 - alpha_bar_t)) * eps_theta)
# 3. Compute the posterior variance \tilde{β}_t = ((1-ᾱ_{t-1})/(1-ᾱ_t)) * β_t.
posterior_var = ((1 - alpha_bar_t_prev) / (1 - alpha_bar_t)) * beta_t
# 4. Add Gaussian noise scaled by √(\tilde{β}_t) unless t == 0.
noise = torch.randn_like(x_t, device=x_t.device) if (t > 0).any() else torch.zeros_like(x_t, device=x_t.device)
mask = (t != 0).float().reshape(-1, *([1] * (x_t.dim() - 1))).to(x_t.device)
sample_prev = mean + mask * torch.sqrt(posterior_var) * noise
# 5. Return the final sample at t-1.
#######################
return sample_prev
```

- Extracts scheduler parameters for the current timestep.
- Calculates the mean and variance for the reverse process.
- Adds noise unless at the final step.
- Returns the denoised sample.

```python
######## TODO ########
if isinstance(t, int):
    t = torch.tensor([t], device=x_t.device)
beta_t = extract(self.betas, t, x_t)
alpha_t = extract(self.alphas, t, x_t)
alpha_bar_t = extract(self.alphas_cumprod, t, x_t)
alpha_bar_t_prev = extract(torch.cat([self.alphas_cumprod.new_ones(1), self.alphas_cumprod[:-1]], 0), t, x_t)
# Posterior mean (from DDPM paper)
mean = (
    torch.sqrt(alpha_bar_t_prev) * beta_t * x0_pred + torch.sqrt(alpha_t) * (1 - alpha_bar_t_prev) * x_t
) / (1 - alpha_bar_t)
posterior_var = ((1 - alpha_bar_t_prev) / (1 - alpha_bar_t)) * beta_t
noise = torch.randn_like(x_t, device=x_t.device) if (t > 0).any() else torch.zeros_like(x_t, device=x_t.device)
mask = (t != 0).float().reshape(-1, *([1] * (x_t.dim() - 1))).to(x_t.device)
sample_prev = mean + mask * torch.sqrt(posterior_var) * noise
#######################
return sample_prev
```

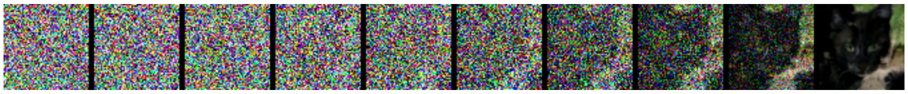
- Uses the predicted clean image to compute the mean for the reverse process.
- Adds noise as in DDPM.

```python
######## TODO ########
if isinstance(t, int):
    t = torch.tensor([t], device=x_t.device)
beta_t = extract(self.betas, t, x_t)
alpha_t = extract(self.alphas, t, x_t)
alpha_bar_t = extract(self.alphas_cumprod, t, x_t)
alpha_bar_t_prev = extract(torch.cat([self.alphas_cumprod.new_ones(1), self.alphas_cumprod[:-1]], 0), t, x_t)
posterior_var = ((1 - alpha_bar_t_prev) / (1 - alpha_bar_t)) * beta_t
noise = torch.randn_like(x_t, device=x_t.device) if (t > 0).any() else torch.zeros_like(x_t, device=x_t.device)
mask = (t != 0).float().reshape(-1, *([1] * (x_t.dim() - 1))).to(x_t.device)
sample_prev = mean_theta + mask * torch.sqrt(posterior_var) * noise
#######################
```

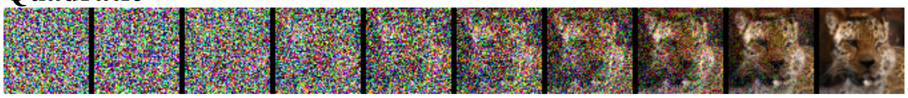
- Implements the reverse step when the network directly predicts the posterior mean.

V. Show the trajectory fig and compare results across the three beta schedulers.

1. **Linear**



2. **Quadratic**

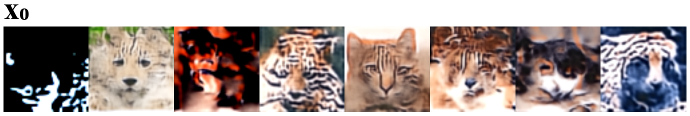


3. **Cosine**

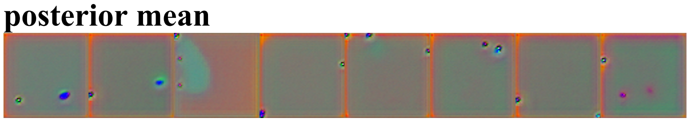
VI. Show the results of different predictors and discuss.

1. **Noise**

2. **x_0**

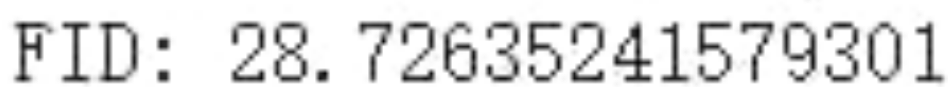


3. **posterior mean**



VII. Screenshot of the Best FID of your training result, explain the training setting.

The best FID score is 28.7 with linear schedulers and x0 predictors



FID: 28.72635241579301

- **Beta Schedule:** Linear (--mode linear)
- **Predictor:** x_0 (--predictor x0)
- **Batch Size:** 16 (--batch_size 16)
- **Image Resolution:** 64x64 (--image_resolution 64)
- **Number of Training Steps:** 50,000 (--train_num_steps 50000)
- **Diffusion Timesteps:** 1,000 (--num_diffusion_train_timesteps 1000)
- **Beta Range:** β_1 = 1e-4, β_T = 0.02 (--beta_1 1e-4 --beta_T 0.02)
- **Seed:** 63 (--seed 63)
- **Optimizer:** Adam, learning rate 2e-4
- **Scheduler:** LambdaLR, warmup steps 200
- **Dataset:** AFHQ, max 3,000 images per category
- **Checkpoint/Results Directory:** Saved to Google Drive (/content/drive/MyDrive/results/predictor_x0/beta_linear/...)
- **Sampling Frequency:** Every 2,000 steps, model samples and saves checkpoint