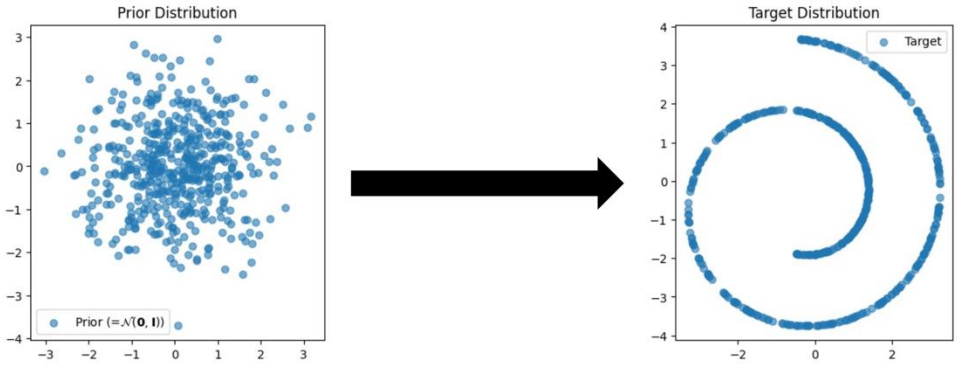
Lab 1 - DDPM



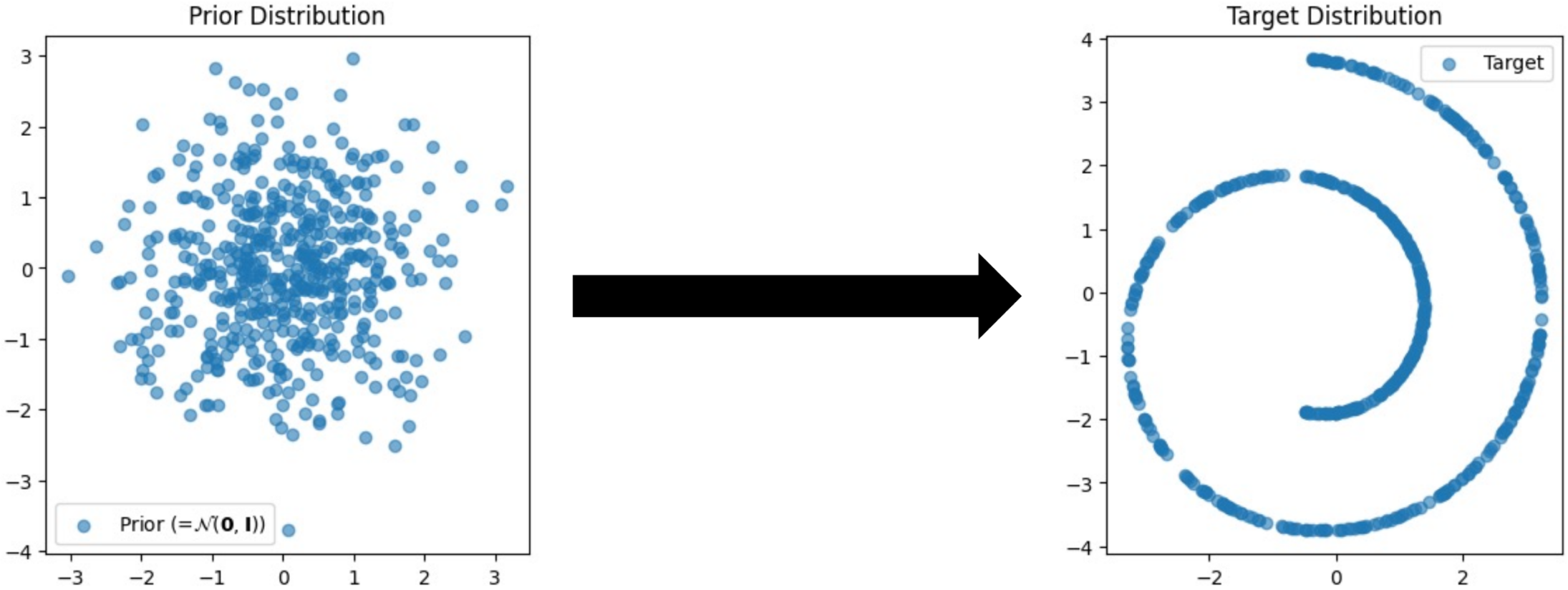
Task 1 - Swiss Roll

Task 2 - Image Generation

DDPM paper

Task 1 - Swiss Roll

Let’s begin by modeling a simple distribution of 2D points (“Swiss Roll”).

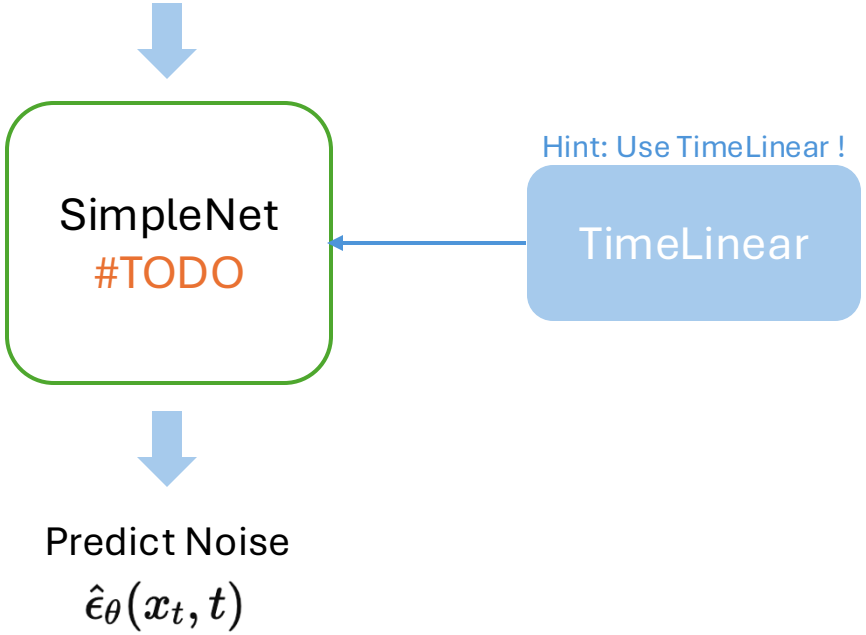


Task 1

#TODO1 - SimpleNet

- 2d_plot_diffusion_todo/network.py

1. Noise data x_t
2. Current timestep t



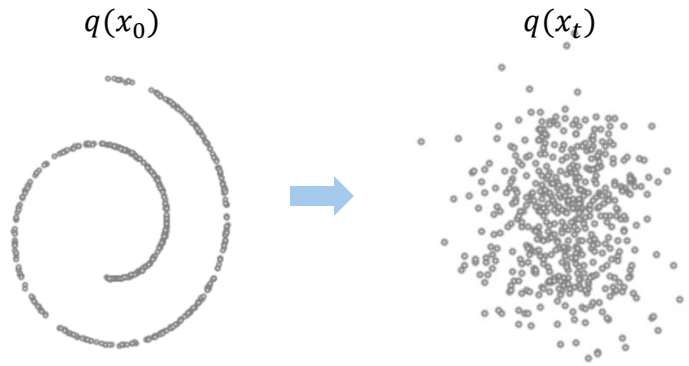
Reverse Process

```python
class SimpleNet(nn.Module):
    def __init__(
        self, dim_in: int, dim_out: int, dim_hids: List[int], num_timesteps: int
    ):
        super().__init__()
        """
        (TODO) Build a noise estimating network.

        Args:
            dim_in: dimension of input
            dim_out: dimension of output
            dim_hids: dimensions of hidden features
            num_timesteps: number of timesteps
        """

        ######## TODO ########
        # DO NOT change the code outside this part.

        ######################

    def forward(self, x: torch.Tensor, t: torch.Tensor):
        """
        (TODO) Implement the forward pass. This should output
        the noise prediction of the noisy input x at timestep t.

        Args:
            x: the noisy data after t period diffusion
            t: the time that the forward diffusion has been running
        """
        ######## TODO ########
        # DO NOT change the code outside this part.

        ######################
        return x
```

Task 1

#TODO2 - q_sample

- 2d_plot_diffusion_todo/ddpm.py



$$q(x_t \mid x_0) = N(x_t; \sqrt{\bar{\alpha}_t}\, x_0, (1 - \bar{\alpha}_t) I)\,.$$

```python
def q_sample(self, x0, t, noise=None):
    """
    sample x_t from q(x_t | x_0) of DDPM.

    Input:
        x0 (`torch.Tensor`): clean data to be mapped to timestep t in the forward process of DDPM.
        t (`torch.Tensor`): timestep
        noise (`torch.Tensor`, optional): random Gaussian noise. if None, randomly sample Gaussian noise i
    Output:
        xt (`torch.Tensor`): noisy samples
    """
    if noise is None:
        noise = torch.randn_like(x0)

    ######## TODO ########
    # DO NOT change the code outside this part.
    # Compute xt.
    alphas_prod_t = extract(self.var_scheduler.alphas_cumprod, t, x0)
    xt = x0

    ######################

    return xt
```

Task 1

Forward Process

#TODO2 - q_sample

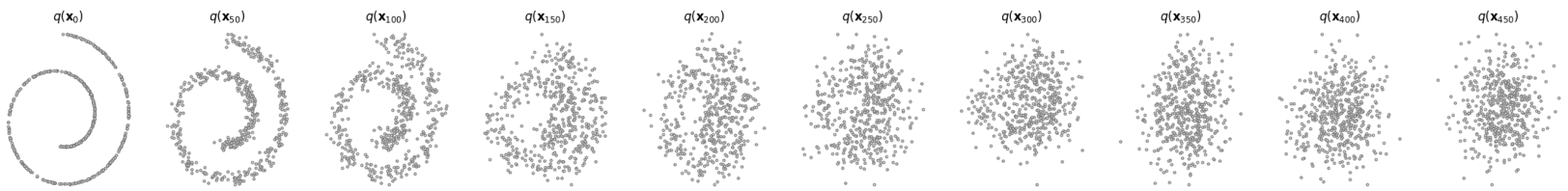
- 2d_plot_diffusion_todo/ddpm_tutorual.py

Check your implementation of q_sample !

Visualize q(x_t)

```python
fig, axs = plt.subplots(1, 10, figsize=(28, 3))
for i, t in enumerate(range(0, 500, 50)):
    x_t = ddpm.q_sample(target_ds[:num_vis_particles].to(device), (torch.ones(num_vis_par
    x_t = x_t.cpu()
    axs[i].scatter(x_t[:,0], x_t[:,1], color='white',edgecolor='gray', s=5)
    axs[i].set_axis_off()
    axs[i].set_title('$q(\mathbf{x}_{'+str(t)+'})$')
```

Python



Task 1

Reverse Process

#TODO3 - p_sample

- 2d_plot_diffusion_todo/ddpm.py

How to get $x_t \xrightarrow{p_\theta} x_{t-1}$?

1. Predict noise from SimpleNet
2. $\mu_\theta(x_t, t) = \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}} \hat{\epsilon}_\theta(x_t, t) \right)$ **Eq. 11**
3. $\tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t} \beta_t$ **Sec. 3.2**
4. $x_{t-1} = \mu_\theta(x_t, t) + \sqrt{\tilde{\beta}_t}\, z, \quad z \sim \mathcal{N}(0, I)$ **Eq. 11**

```python
def p_sample(self, xt, t):
    """
    One step denoising function of DDPM: x_t -> x_{t-1}.
    Input:
        xt (`torch.Tensor`): samples at arbitrary timestep t.
        t (`torch.Tensor`): current timestep in a reverse process.
    Ouptut:
        x_t_prev (`torch.Tensor`): one step denoised sample. (= x_{t-1})
    """
    ######## TODO ########
    # DO NOT change the code outside this part.
    # compute x_t_prev.
    if isinstance(t, int):
        t = torch.tensor([t]).to(self.device)
    eps_factor = (1 - extract(self.var_scheduler.alphas, t, xt)) / (
        1 - extract(self.var_scheduler.alphas_cumprod, t, xt)
    ).sqrt()

    beta_t      = extract(self.var_scheduler.betas,          t, xt)        # β_t
    alpha_t     = extract(self.var_scheduler.alphas,         t, xt)        # α_t =
    alpha_bar_t = extract(self.var_scheduler.alphas_cumprod, t, xt)        # \bar{
    t_prev      = (t - 1).clamp(min=0)
    alpha_bar_t_prev = extract(self.var_scheduler.alphas_cumprod, t_prev, xt) # \bar

    # 1. predict noise

    # 2. Posterior mean

    # 3. Posterior variance

    # 4. Reverse step

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

Task 1

#TODO4 - p_sample loop

- 2d_plot_diffusion_todo/ddpm.py

```python
def p_sample_loop(self, shape):
    """
    The loop of the reverse process of DDPM.

    Input:
        shape (`Tuple`): The shape of output. e.g., (num particles, 2)
    Output:
        x0_pred (`torch.Tensor`): The final denoised output through the DDPM reverse process.
    """
    ######## TODO ########
    # DO NOT change the code outside this part.
    # sample x0 based on Algorithm 2 of DDPM paper.
    x0_pred = torch.zeros(shape).to(self.device)

    ######################
    return x0_pred
```

Task 1

#TODO5 - compute_loss

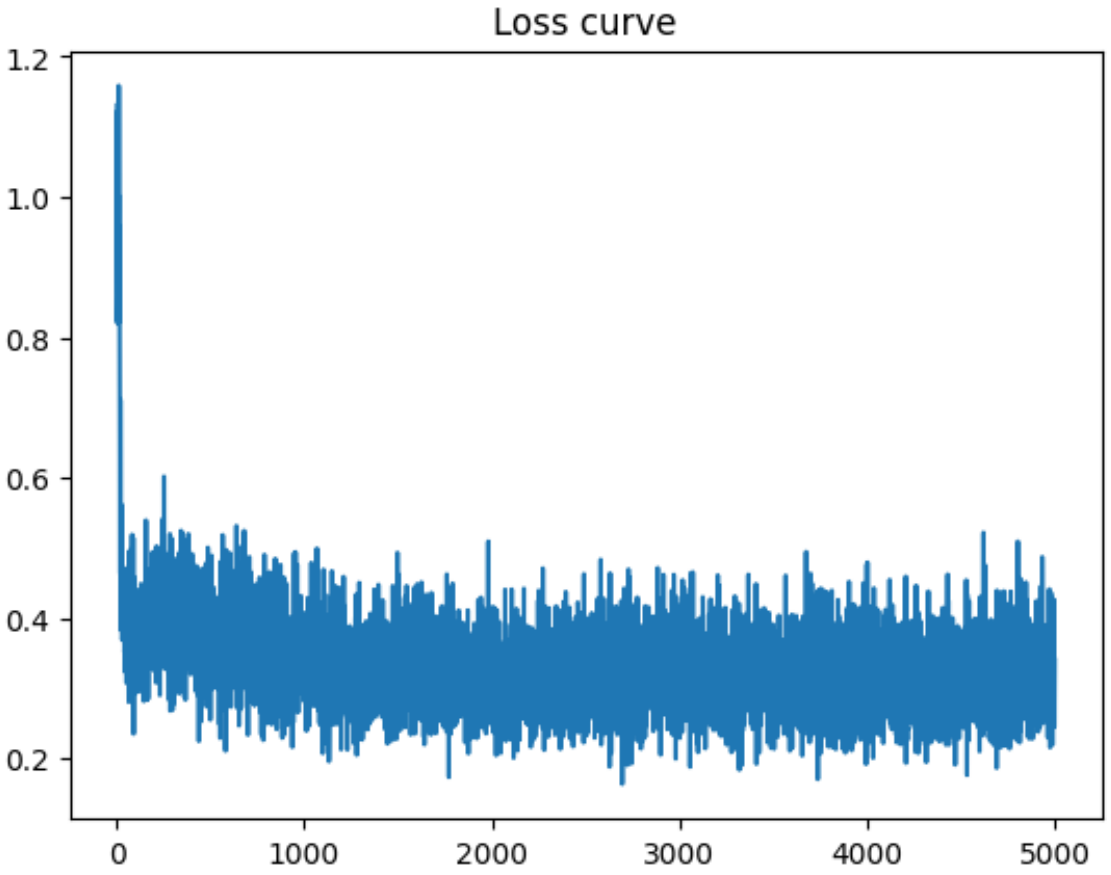
- 2d_plot_diffusion_todo/ddpm.py

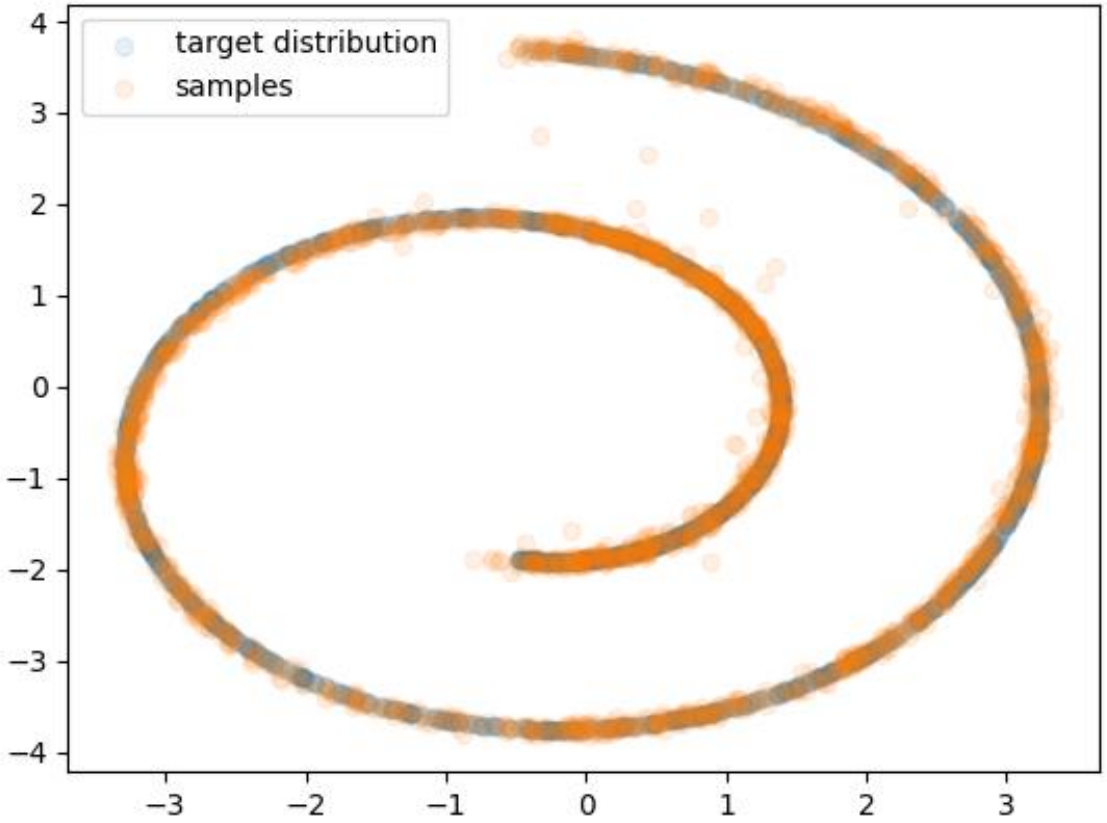
```python
def compute_loss(self, x0):
    """
    The simplified noise matching loss corresponding Equation 14 in DDPM paper.
    Input:
        x0 (`torch.Tensor`): clean data
    Output:
        loss: the computed loss to be backpropagated.
    """
    ######## TODO ########
    # DO NOT change the 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

    # 3) predict noise

    # 4) MSE loss (eps, eps_pred)

    loss = None

    ######################
    return loss
```

Task 1

#TODO5 – train & evaluate

- 2d_plot_diffusion_todo/ddpm_tutorial.py

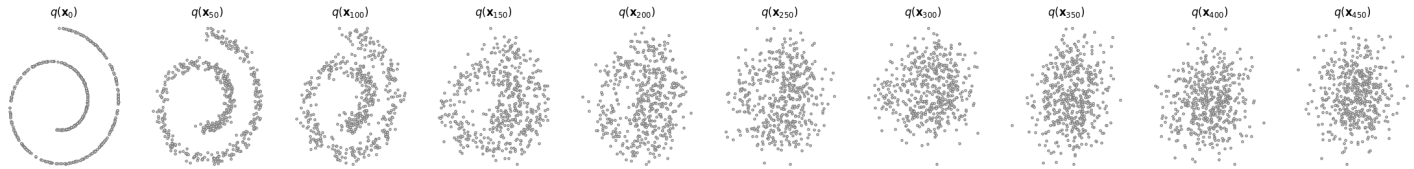




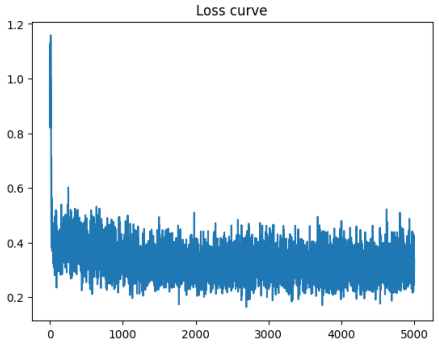
Task 1

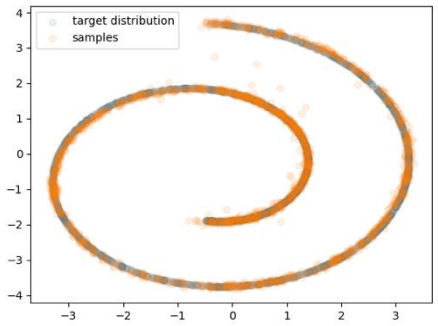
#Report

1. Complete and Explain of all **#TODO** code (20pts)
2. Fig of your implementation of q_sample (10pts)



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





Task 2 – Image Generation

What to Do: Task 2

Samples from our model trained using the AFHQ dataset.

Bring your codes from Task 1!

The code needs to be modified, but the changes should be kept minimal.

- q_sample ➡ add_noise;
- p_sample ➡ step;
- compute_loss ➡ get_loss.

Task 2

Forward Process

TODO1 – add_noise

- image_diffusion_todo/scheduler.py

```python
def add_noise(
    self,
    x_0: torch.Tensor,
    t: torch.IntTensor,
    eps: Optional[torch.Tensor] = None,
):
    """
    A forward pass of a Markov chain, i.e., q(x_t | x_0).

    Input:
        x_0 (`torch.Tensor [B,C,H,W]`): samples from a real data distribution q(x_0).
        t: (`torch.IntTensor [B]`)
        eps: (`torch.Tensor [B,C,H,W]`, optional): if None, randomly sample Gaussian noise in the function.
    Output:
        x_t: (`torch.Tensor [B,C,H,W]`): noisy samples at timestep t.
        eps: (`torch.Tensor [B,C,H,W]`): injected noise.
    """

    if eps is None:
        eps       = torch.randn(x_0.shape, device='cuda')

    ######## TODO ########
    # DO NOT change the code outside this part.
    # Assignment 1. Implement the DDPM forward step.
    x_t = None


    ######################

    return x_t, eps
```

Hint : Refer to 2d_plot_diffusion_todo/ddpm.py – **q_sample**

Task 2

TODO4 – beta scheduling

- image_diffusion_todo/scheduler.py

We know

$$q(x_t \mid x_0) = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t}\, \epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

and

$$\alpha_t = 1 - \beta_t$$

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

Implement the **cosine scheduler**.

```python
if mode == "linear":
    betas = torch.linspace(beta_1, beta_T, steps=num_train_timesteps)
elif mode == "quad":
    betas = (
        torch.linspace(beta_1**0.5, beta_T**0.5, num_train_timesteps) ** 2
    )
elif mode == "cosine":
    ######## 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].
    raise NotImplementedError("TODO: Implement cosine beta schedule here!")
```

Task 2

TODO5 – predictor

- image_diffusion_todo/scheduler.py

Implement the **reverse step** according to the chosen **predictor** (noise, x_0, or mean).

```python
def step_predict_noise(self, x_t: torch.Tensor, t: int, eps_theta: torch.Tensor):
    """
    Noise prediction version (the standard DDPM formulation).

    Input:
        x_t: noisy image at timestep t
        t: current timestep
        eps_theta: predicted noise ε̂_θ(x_t, t)
    Output:
        sample_prev: denoised image sample at timestep t-1
    """
    ######## TODO ########
    # 1. Extract beta_t, alpha_t, and alpha_bar_t from the scheduler.
    # 2. Compute the predicted mean μ_θ(x_t, t) = 1/√α_t * (x_t - (β_t/√(1-ᾱ_t)) * ε
    # 3. Compute the posterior variance \tilde{β}_t = ((1-ᾱ_{t-1})/(1-ᾱ_t)) * β_t.
    # 4. Add Gaussian noise scaled by √(\tilde{β}_t) unless t == 0.
    # 5. Return the final sample at t-1.
    sample_prev = None
    ########################
    return sample_prev
```

```python
def step_predict_x0(self, x_t: torch.Tensor, t: int, x0_pred: torch.Tensor):
    """
    x0 prediction version (alternative DDPM objective).

    Input:
        x_t: noisy image at timestep t
        t: current timestep
        x0_pred: predicted clean image x̂₀(x_t, t)
    Output:
        sample_prev: denoised image sample at timestep t-1
    """
    ######## TODO ########

    sample_prev = None
    ########################
    return sample_prev


def step_predict_mean(self, x_t: torch.Tensor, t: int, mean_theta: torch.Tensor):
    """
    Mean prediction version (directly outputting the posterior mean).

    Input:
        x_t: noisy image at timestep t
        t: current timestep
        mean_theta: network-predicted posterior mean μ̂_θ(x_t, t)
    Output:
        sample_prev: denoised image sample at timestep t-1
    """
    ######## TODO ########

    sample_prev = None
    ########################
    return sample_prev
```

Task 2

TODO5 – predictor

- image_diffusion_todo/model.py

Implement **loss functions** for training the network
to predict different targets: **noise**, x_0, or the **posterior mean**.

```python
def get_loss(self, x0, class_label=None, noise=None):
    if self.predictor == "noise":
        return self.get_loss_noise(x0, class_label, noise)
    elif self.predictor == "x0": #### TODO
        return self.get_loss_x0(x0, class_label, noise)
    elif self.predictor == "mean": #### TODO
        return self.get_loss_mean(x0, class_label, noise)
    else:
        raise ValueError(f"Unknown predictor: {self.predictor}")
```

```python
def get_loss_x0(self, x0, class_label=None, noise=None):
    ######## TODO ########
    # Here we implement the "predict x0" version.
    # 1. Sample a timestep and add noise to get (x_t, noise).
    # 2. Pass (x_t, timestep) into self.network, where the output should represent the clean sample x0_pred.
    # 3. Compute the loss as MSE(predicted x0_pred, ground-truth x0).
    ######################
    loss = None
    return loss

def get_loss_mean(self, x0, class_label=None, noise=None):
    ######## TODO ########
    # Here we implement the "predict mean" version.
    # 1. Sample a timestep and add noise to get (x_t, noise).
    # 2. Pass (x_t, timestep) into self.network, where the output should represent the posterior mean μθ(x_t, t)
    # 3. Compute the *true* posterior mean from the closed-form DDPM formula (using x0, x_t, noise, and schedule
    # 4. Compute the loss as MSE(predicted mean, true mean).
    ######################
    loss = None
    return loss
```

Task 2

TODO6 - train

- image_diffusion_todo/train.py

python train.py --mode {BETA_SCHEDULING} --predictor {PREDICTOR}

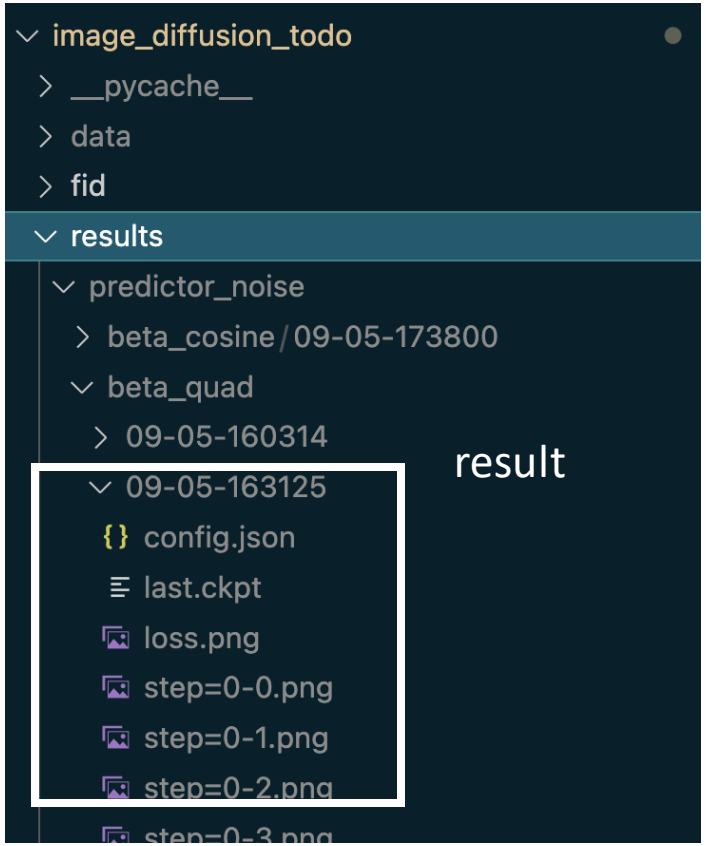
Try different **beta scheduling**:

1. linear
2. quadratic
3. cosine

Try different **predictor**:

1. noise (default)
2. mean
3. x_0

Please train for at least **50,000–100,000** iterations.
Training may take **over 6 hours per time**, so start ASAP!

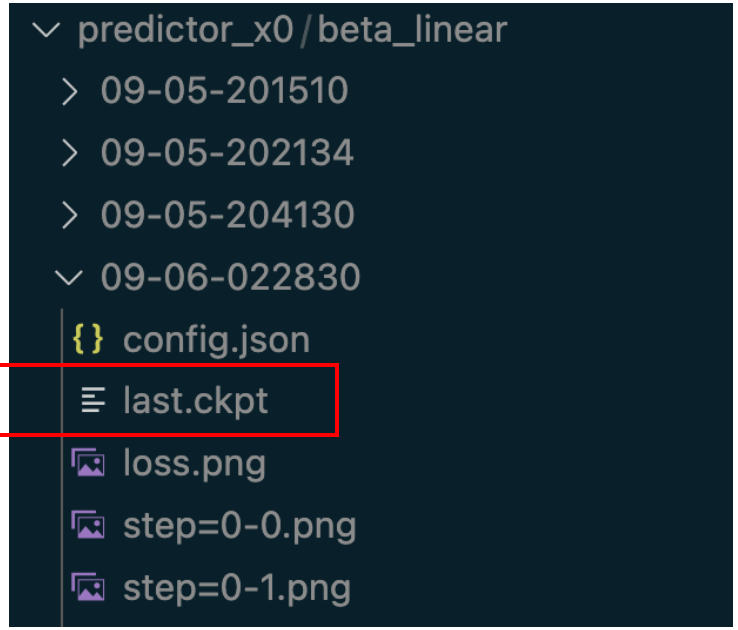


Task 2

 # TODO7 - sampling

- image_diffusion_todo/sampling.py

**python sampling.py --ckpt_path {CKPT} --save_dir {SAMPLING_SAVE_DIR} **
--mode {BETA_SCHEDULING} --predictor {PREDICTOR}



Task 2

 # TODO8 - evaluate

- image_diffusion_todo/dataset.py

Prepare the data for evaluation by running

python dataset.py (Only once!)

This will create the eval directory under data/afhq.

Do NOT forget to run this. Otherwise, you will get incorrect FIDs!

cd fid

python measure_fid.py ../data/afhq/eval {SAMPLING_SAVE_DIR}

```
FID: 229.33834138594412
```

Incorrect FIDs

```
FID: 10.844529455220403
```

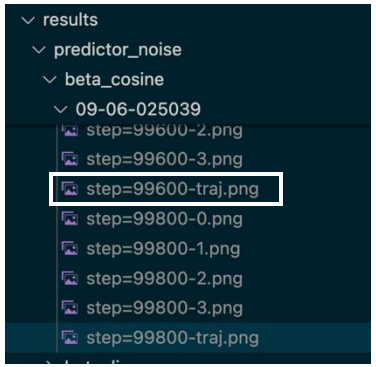
Correct FIDs

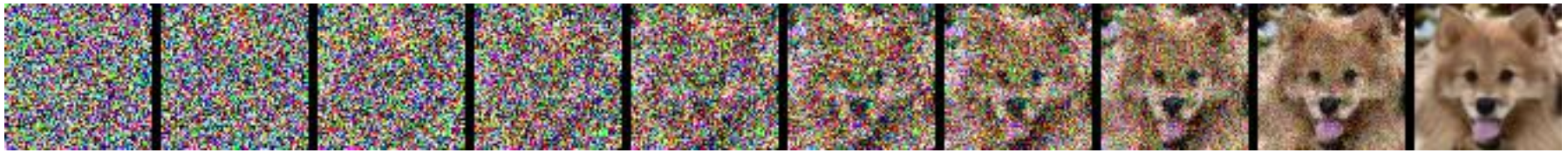
Task 2

#Report

1. Complete and Explain of all **#TODO** code. (30pts)
2. Show the **trajectory fig** and compare results across the three **beta schedulers**.

   (linear, quadratic, cosine) (10pts)





3. Show the results of different **predictors** **(noise, x_0, posterior mean)** and discuss. (10pts)

8 samples per predictor, 24 in total.

4. Screenshot of the **Best FID** of your training result, explain the training setting. (20pts)

- 20 pts: FID < 15
- 15 pts: 15 ≤ FID < 20
- 10 pts: 20 ≤ FID < 30
- 5 pts: 30 ≤ FID < 40
- 0 pts: FID ≥ 40

Submit

Create a single ZIP file {ID}_lab1.zip including:

- The **PDF** file formatted following the guideline;
- Your **code** without checkpoints for DDPMs and the Inception Network

412551014_lab1.zip

- report.pdf

- 2d_plot_diffusion_todo

- image_diffusion_todo

Thank you