CS492D: Diffusion Models and Their Applications

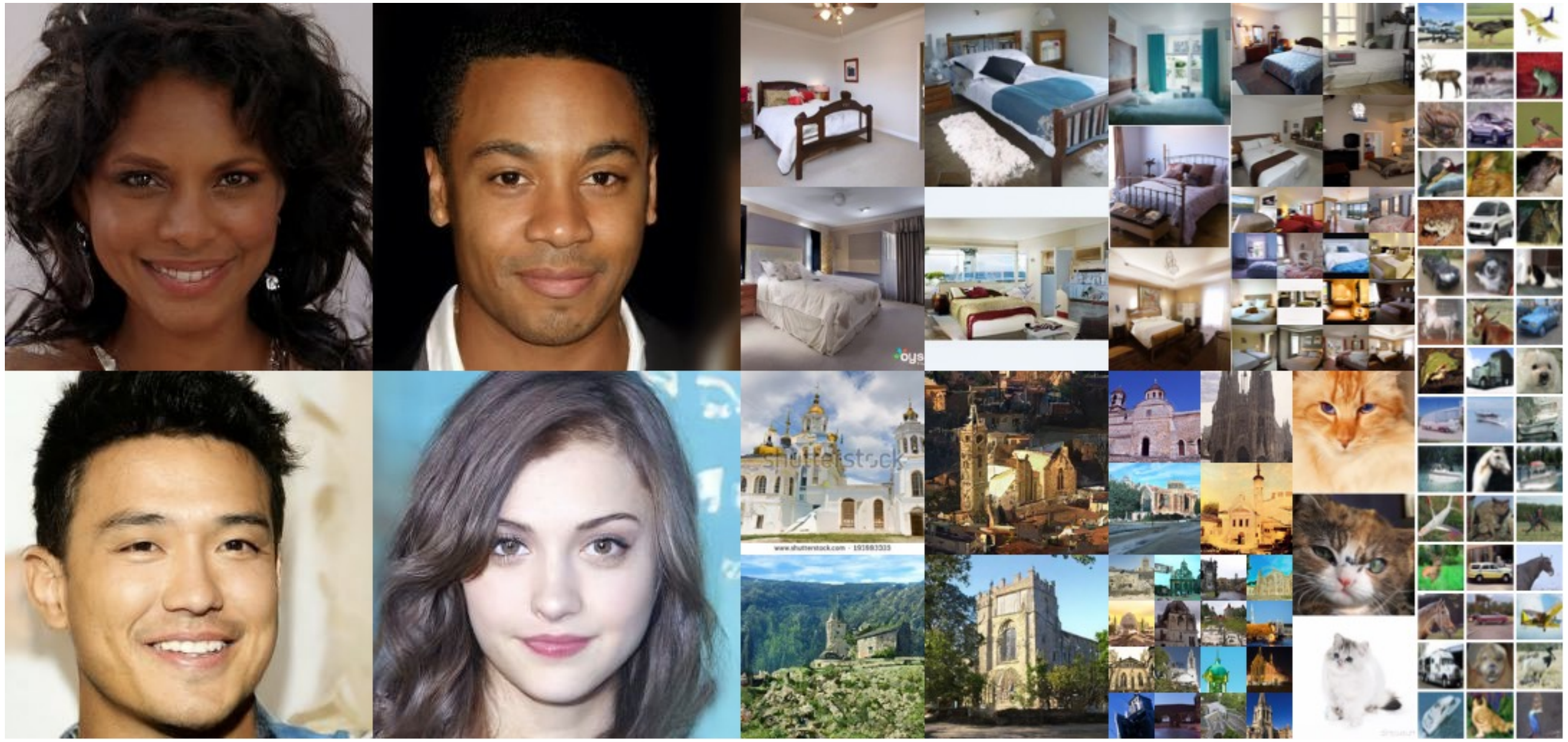
Assignment 1 Session

Seungwoo Yoo

Fall 2024
KAIST

Introduction

In Assignment 1, you will implement the key components of **Denoising Diffusion Probabilistic Models (DDPMs)**.

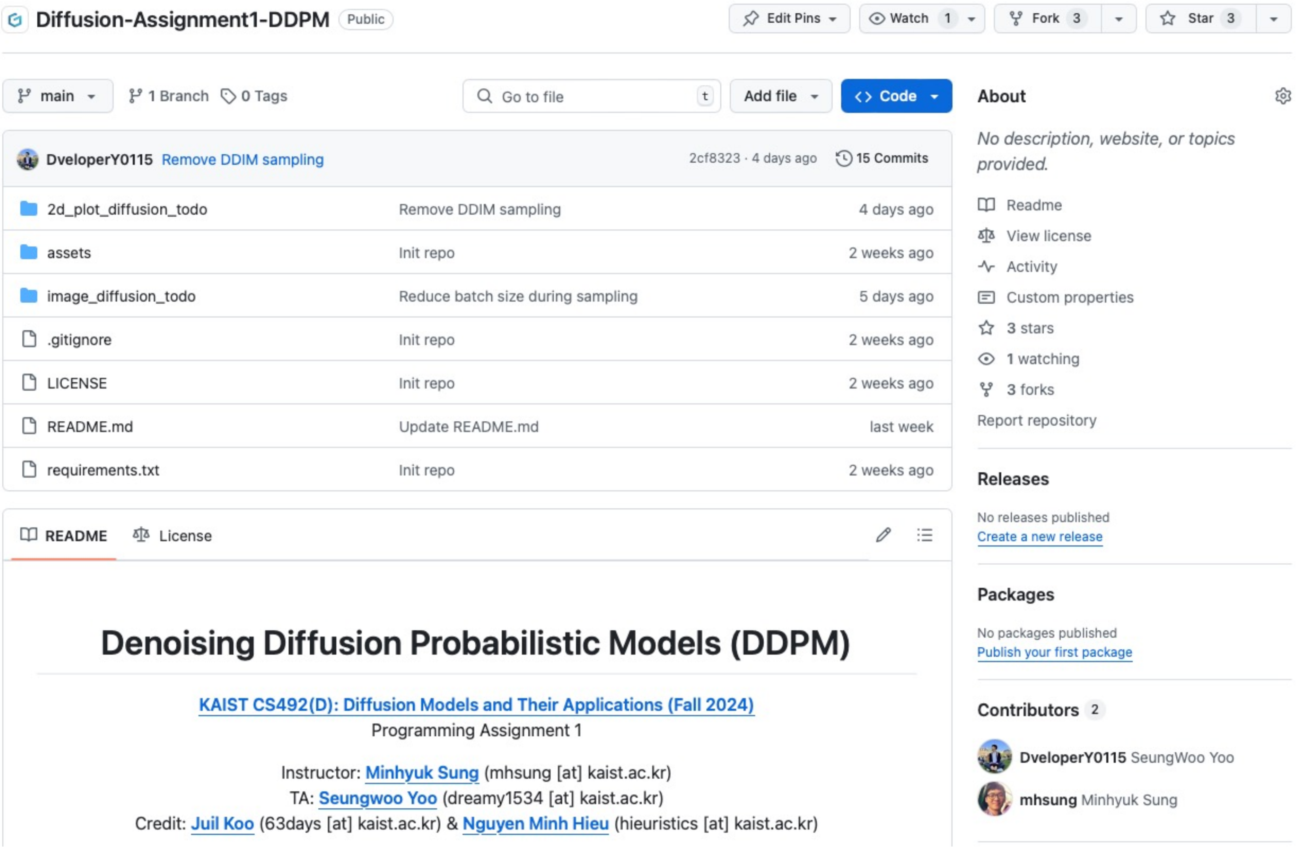


Denoising Diffusion Probabilistic Models, Ho *et al.*, NeurIPS 2020

Introduction

The skeleton code and instructions are available at:

https://github.com/KAIST-Visual-AI-Group/Diffusion-Assignment1-DDPM



Introduction

- All programming assignments are due **two weeks** after the assignment session.
- Late submission will incur **20% penalty** for **each** late day!
- Please carefully check the README of each assignment.

What to Do: Overview

You need to implement three major components of DDPMs:

- Forward Process: $q(\boldsymbol{x}_t|\boldsymbol{x}_0)$
- Reverse Process: $p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$
- Training Objective: $||\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)||^2$

Is it really that simple…?

What to Do: Overview

Yes! Even cutting-edge diffusion models are built this way.

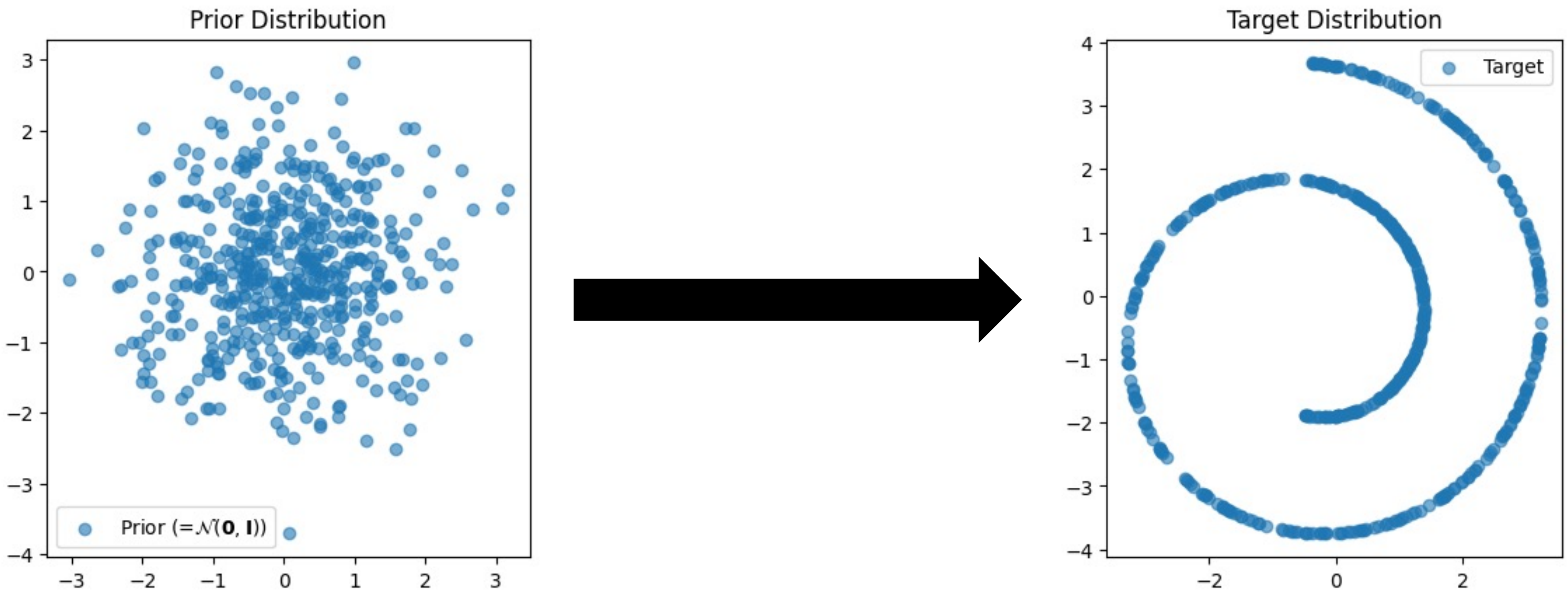
By understanding this basic structure, you can begin exploring more advanced models.

Diffusers, HuggingFace

Stable Diffusion 3, StabilityAI

What to Do: Task 1

Let's begin by modeling a simple distribution of 2D points ("Swiss Roll").



What to Do: Task 1

Design a network that takes

- Noisy data $\boldsymbol{x}_t$;
- Current diffusion timestep t.

Hint: Use the `TimeLinear` class.

```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
```

`2d_plot_diffusion_todo/network.py`

What to Do: Task 1

Implement functions

- q_sample;
- p_sample;
- p_sample_loop;
- compute_loss.

```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 in the function.
    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
```

2d_plot_diffusion_todo/ddpm.py

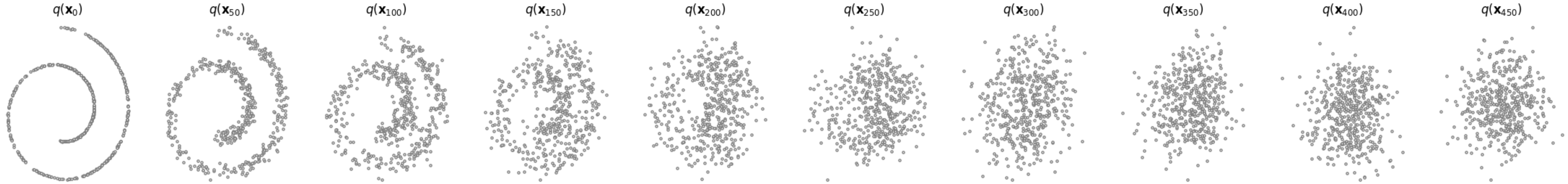
What to Do: Task 1

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_particles) * t).to(device))
    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('$q(\mathbf{x}_{'+str(t)+'})$')
```

[3] Python



`2d_plot_diffusion_todo/ddpm_tutorial.ipynb`

What to Do: Task 1

Implement functions

- q_sample;
- p_sample;
- p_sample_loop;
- compute_loss.

```python
@torch.no_grad()
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()
    eps_theta = self.network(xt, t)

    x_t_prev = xt

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

2d_plot_diffusion_todo/ddpm.py

What to Do: Task 1

Implement functions

- q_sample;
- p_sample;
- **p_sample_loop;**
- compute_loss.

```python
@torch.no_grad()
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
```

2d_plot_diffusion_todo/ddpm.py

What to Do: Task 1

Implement functions

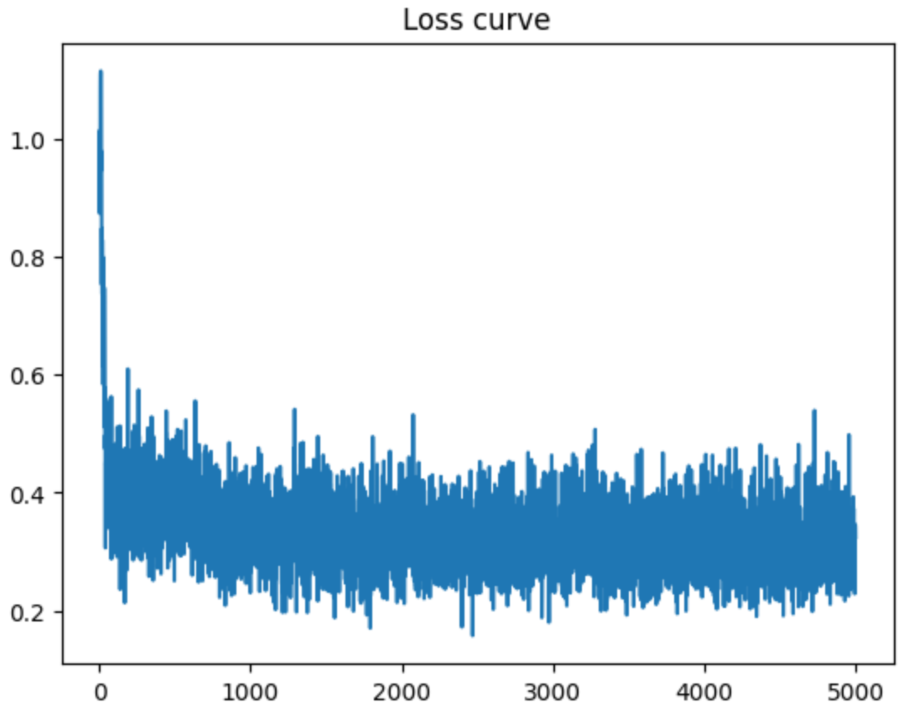
- `q_sample;`
- `p_sample;`
- `p_sample_loop;`
- `compute_loss.`

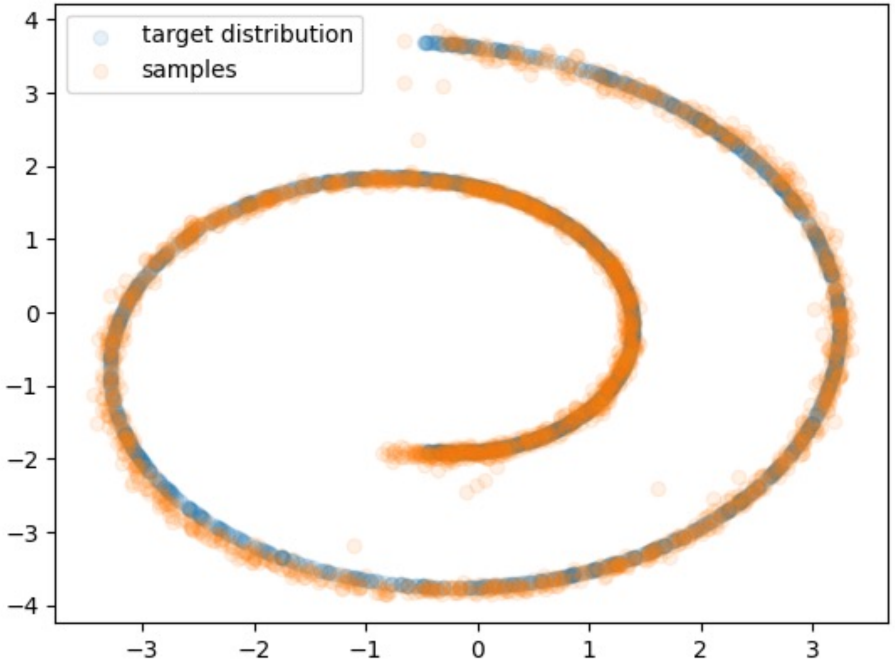
```python
def compute_loss(self, x0):
    """
    The simplified noise matching loss corresponding Equation 14 in DDPM paper.

    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]
    t = (
        torch.randint(0, self.var_scheduler.num_train_timesteps, size=(batch_size,))
        .to(x0.device)
        .long()
    )

    loss = x0.mean()

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

`2d_plot_diffusion_todo/ddpm.py`

What to Do: Task 1

Train your model and observe how the generated samples and the loss curve evolve over time.



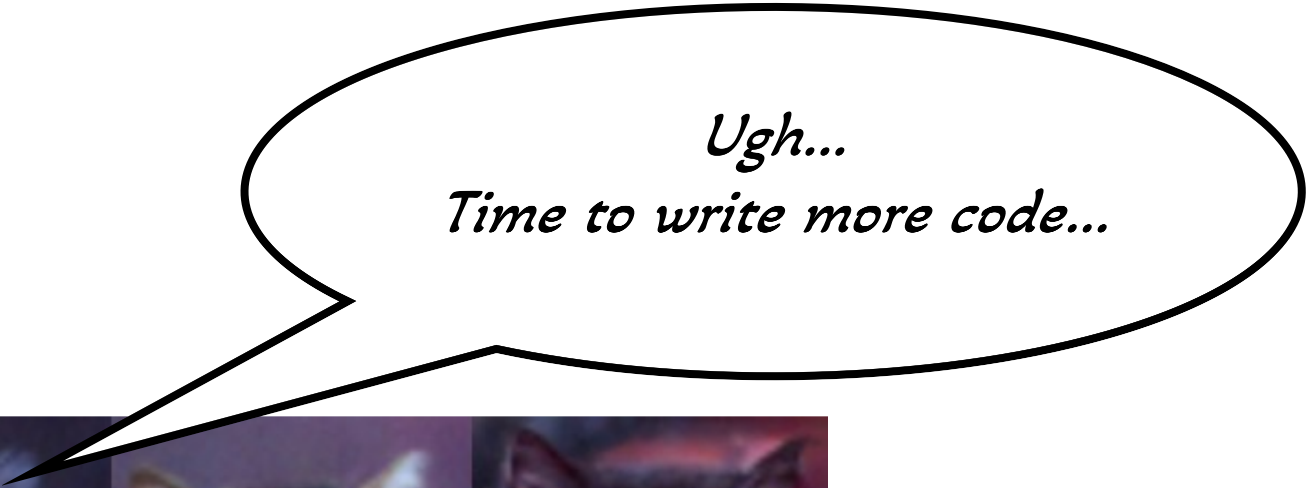


What to Do: Task 2

We will now move on to a more interesting example: image generation.

Samples from our model trained using the AFHQ dataset.

What to Do: Task 2



Samples from our model trained using the AFHQ dataset.

What to Do: Task 2

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.`

What to Do: Task 2

```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 in the function.
    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
```

`2d_plot_diffusion_todo/ddpm.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
```

`image_diffusion_todo/scheduler.py`

What to Do: Task 2

```python
@torch.no_grad()
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()
    eps_theta = self.network(xt, t)

    x_t_prev = xt

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

2d_plot_diffusion_todo/ddpm.py

```python
def step(self, x_t: torch.Tensor, t: int, eps_theta: torch.Tensor):
    """
    One step denoising function of DDPM: x_t → x_{t-1}.

    Input:
        x_t (`torch.Tensor [B,C,H,W]`): samples at arbitrary timestep t.
        t (`int`): current timestep in a reverse process.
        eps_theta (`torch.Tensor [B,C,H,W]`): predicted noise from a learned model.
    Ouptut:
        sample_prev (`torch.Tensor [B,C,H,W]`): one step denoised sample. (= x_{t-1})
    """

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

    return sample_prev
```

image_diffusion_todo/scheduler.py

What to Do: Task 2

```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]
    t = (
        torch.randint(0, self.var_scheduler.num_train_timesteps, size=(batch_size,))
        .to(x0.device)
        .long()
    )

    loss = x0.mean()

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

2d_plot_diffusion_todo/ddpm.py

```python
def get_loss(self, x0, class_label=None, noise=None):
    ######## TODO ########
    # DO NOT change the code outside this part.
    # compute noise matching loss.
    B = x0.shape[0]
    timestep = self.var_scheduler.uniform_sample_t(B, self.device)
    loss = x0.mean()
    ######################
    return loss
```

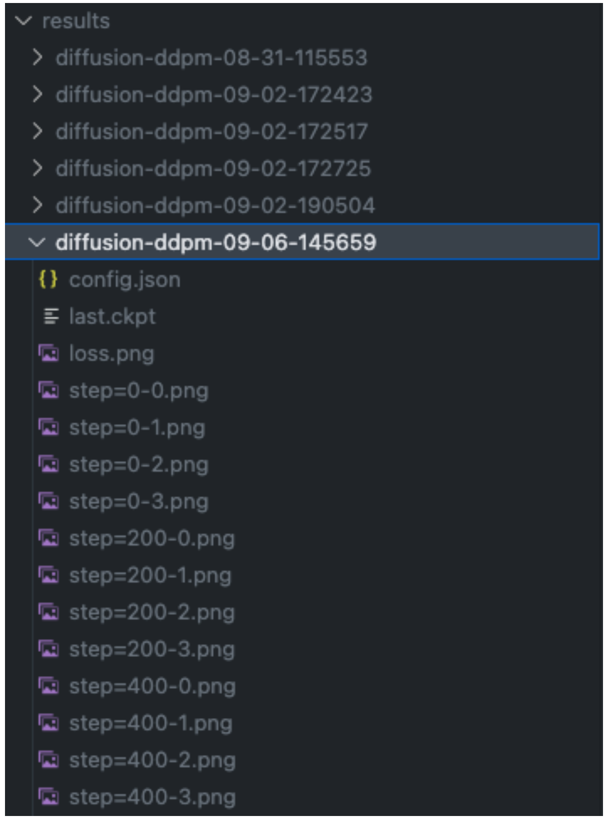
image_diffusion_todo/scheduler.py

What to Do: Task 2

After implementing the functions, start training the model by running

```
python train.py
```

The results will be saved under `results` directory.



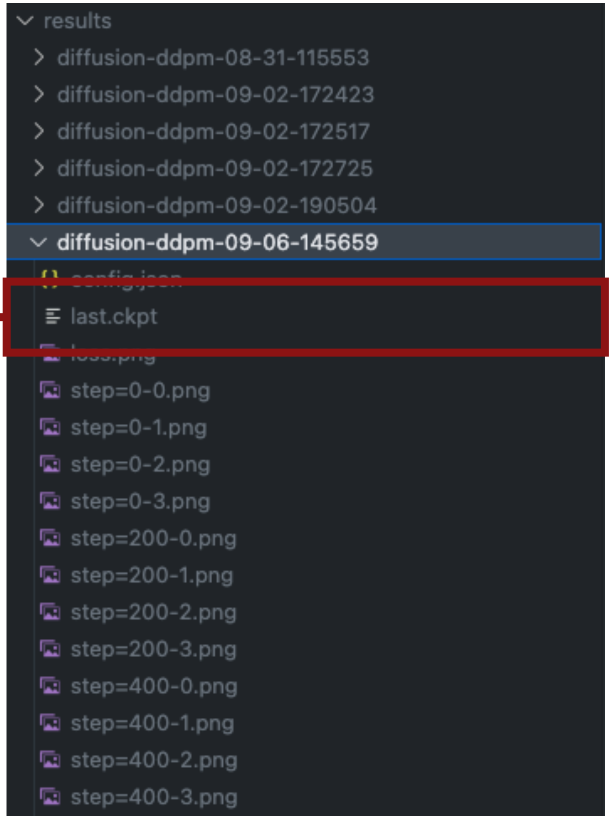
Step 0 | Step 1K | Step 25K | Step 50K | Step 100K

Images logged during training.

What to Do: Task 2

Generate the images using the trained model by running

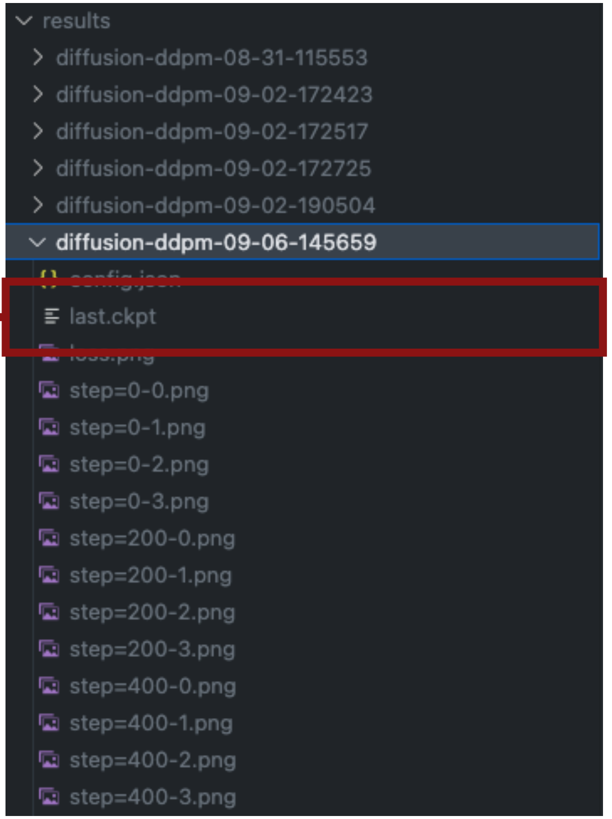
```
python sampling.py \
--ckpt_path {CKPT} --save_dir {SAVE}
```



What to Do: Task 2

Generate the images using the trained model by running

```
python sampling.py \
--ckpt_path {CKPT} --save_dir {SAVE}
```



Samples generated using our model.

What to Do: Task 2

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!

`FID: 229.33834138594412`   `FID: 10.844529455220403`

FID scores across different test sets using the same generated samples.

What to Submit

Compile the following items into a PDF file: `{NAME}_{ID}.pdf`.

Task 1

- A screenshot of the loss curve;
- A screenshot of the Chamfer Distance;
- A visualization of samples generated using your DDPM.

Task 2

- A screenshot of the computed FID;
- At least 8 images generated using your DDPM.

What to Submit

Create a single ZIP file `{NAME}_{ID}.zip` including:

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

Your score will be deducted by 10% for each missing item.

Please check carefully!

Grading

You will receive up to 20 points from this assignment.

Task 1

- 10 points: Achieve CD lower than 20.
- 5 points: Achieve CD greater, or equal to 20 and less than 40.
- 0 point: Otherwise.

Grading

You will receive up to 20 points from this assignment.

Task 2

- 10 points: Achieve FID lower than 20.
- 5 points: Achieve FID greater, or equal to 20 and less than 40.
- 0 point: Otherwise.

Demo

Thank You