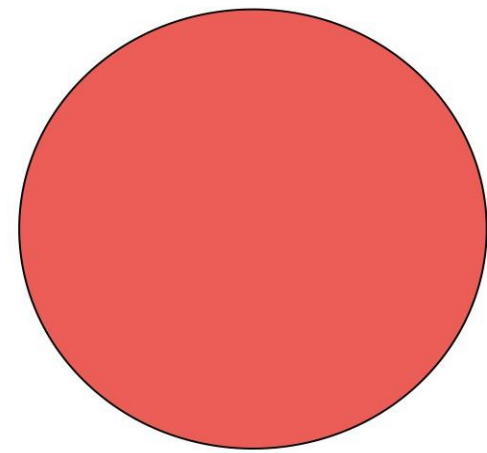


Diffusion Models

Sampling from easy distribution

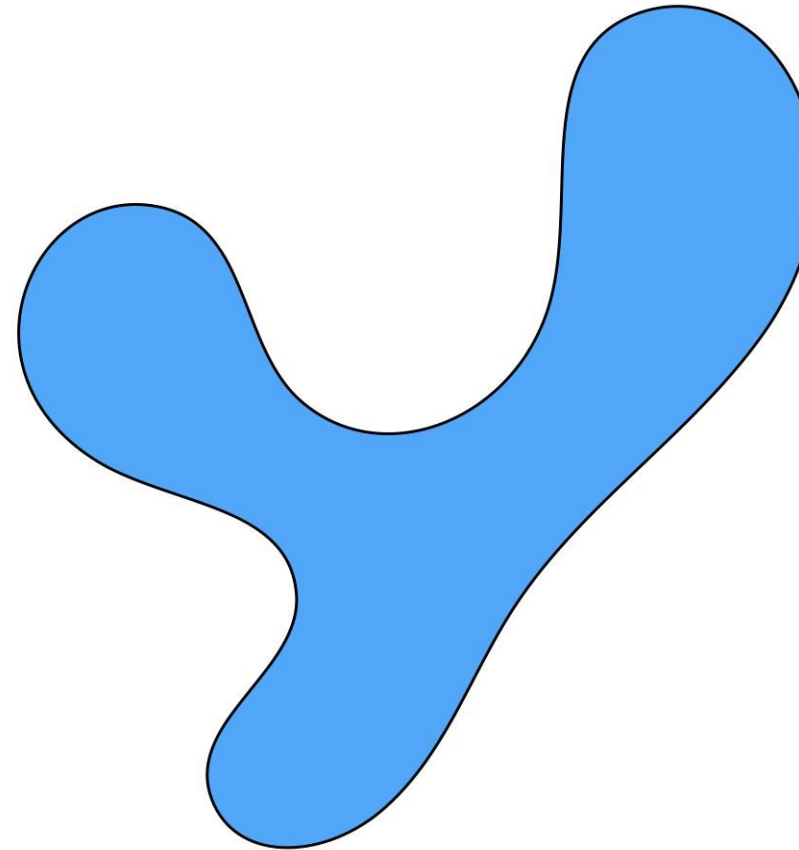
Source distribution



$$p(z)$$

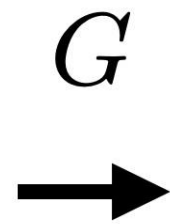
Example : Gaussian

Target distribution



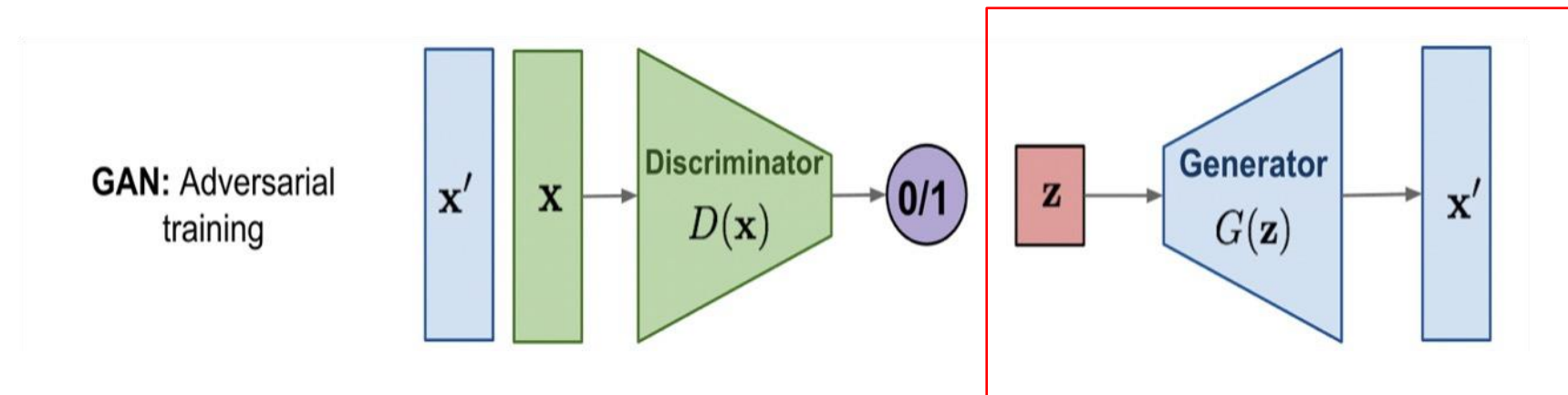
$$p(x)$$

Example : Dogs images



Example GANs

- Examples of a generative model that convert latent noise vector to target distribution in one step are GANs:



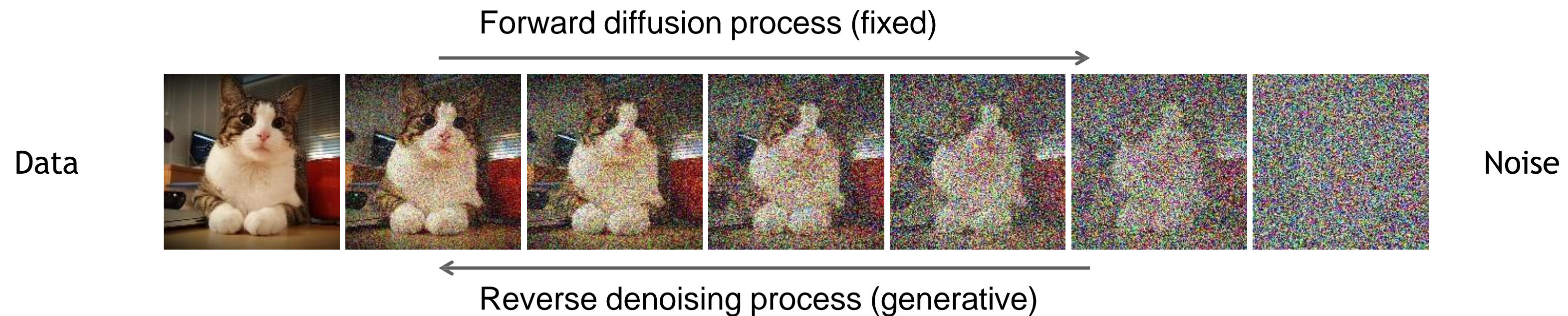
- Training GANs presents several challenges:
 - 1. **Vanishing Gradients:** If the discriminator becomes too proficient, the gradients for the generator can diminish to zero, hindering its learning process.
 - 2. **Mode Collapse:** When the generator finds a highly plausible output, it might start producing only that output. The discriminator, in turn, learns to always reject this output, leading the generator to produce the same output repeatedly, resulting in a negative feedback loop.

Diffusion Main Idea

Core Concept: Diffusion models are a class of generative models that learn to generate data by iteratively denoising from a noise distribution. They are based on two processes:

1.Forward Diffusion Process: Gradually add noise to the data over several steps until it becomes pure noise.

2.Reverse Diffusion Process: Learn to reverse the noise addition process, step-by-step, to generate new data from the noise.



Markov Chain: Main Idea

Definition:

- A Markov Chain is a mathematical system that transitions from one state to another within a finite or countable state space.
- It is characterized by the property that the future state depends only on the current state and not on the sequence of events that preceded it.

Key Properties:

- **State Space:** Set of all possible states.
- **Transition Probability:** Probability of moving from one state to another.
- **Memoryless (Markov Property):** The next state depends only on the current state.

Diffusion Models as Markov Chains

Core Concept:

- Diffusion models can be viewed as Markov chains where each step involves a transition from one state to another by adding or removing noise.

How It Works:

1. Forward Diffusion (Markov Chain):

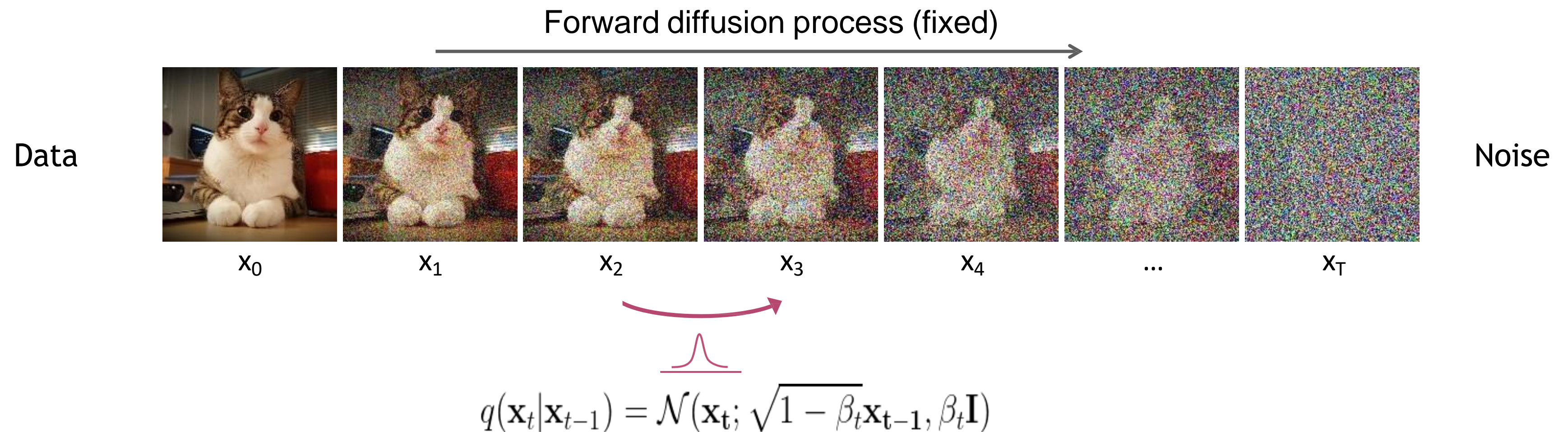
1. Start with the original data.
2. Add small amounts of Gaussian noise at each step.
3. Each step is a state transition in a Markov chain, where the state at step $t+1$ depends only on the state at step t .

2. Reverse Diffusion (Markov Chain):

1. Begin with pure noise.
2. Iteratively denoise the data by reversing the noise addition process.
3. Each denoising step is a state transition in a Markov chain, where the state at step $t-1$ depends only on the state at step t .

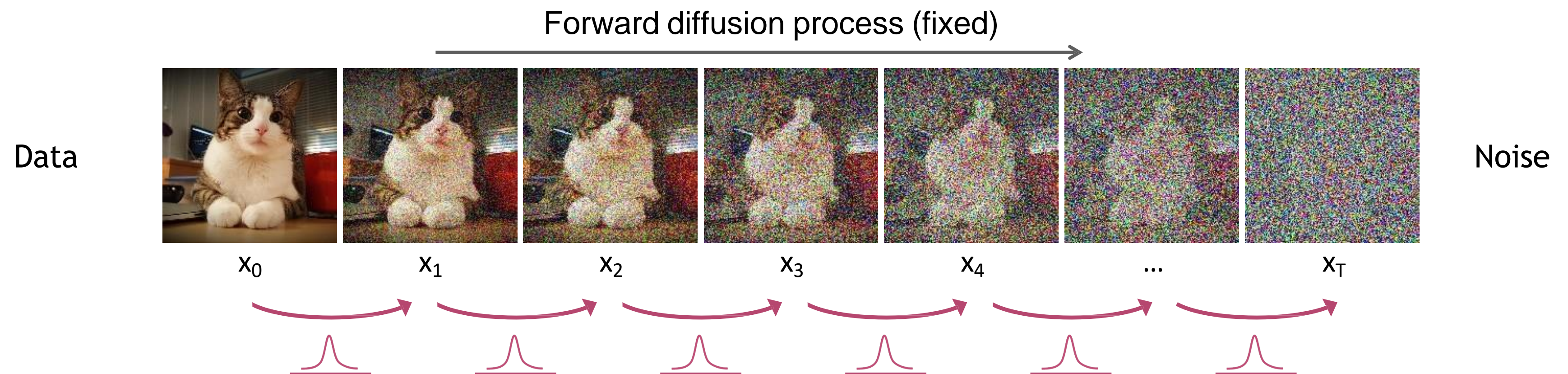
The Forward Diffusion Process

The formal definition of the forward process in T steps:



The Forward Diffusion Process

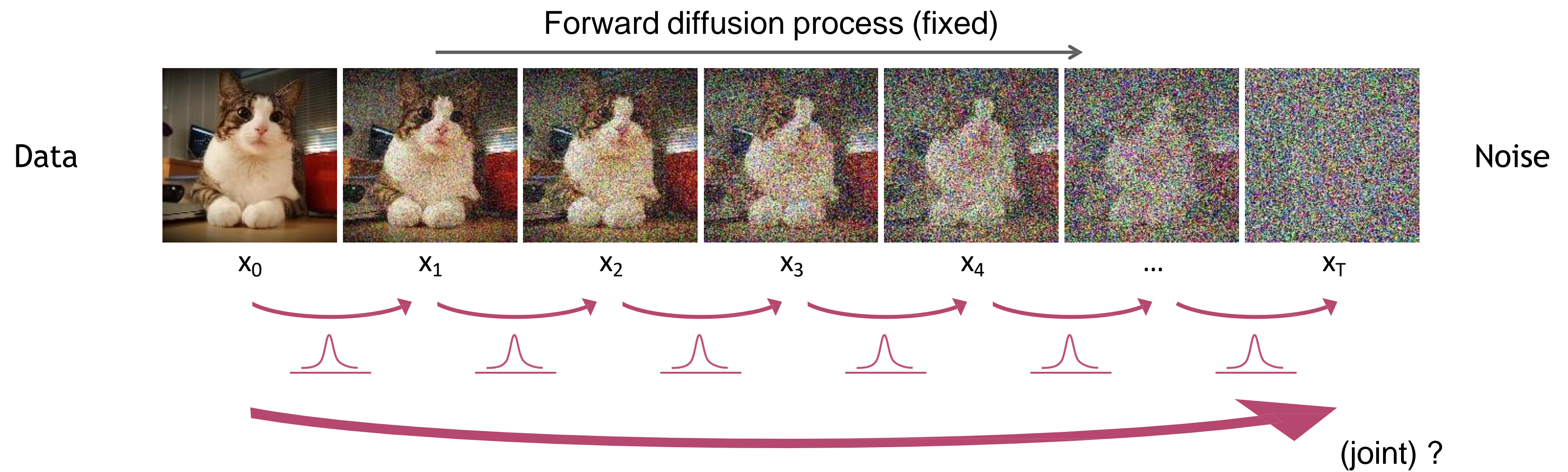
The formal definition of the forward process in T steps:



$$q(\mathbf{x}_t | \mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1 - \beta_t} \mathbf{x}_{t-1}, \beta_t \mathbf{I})$$

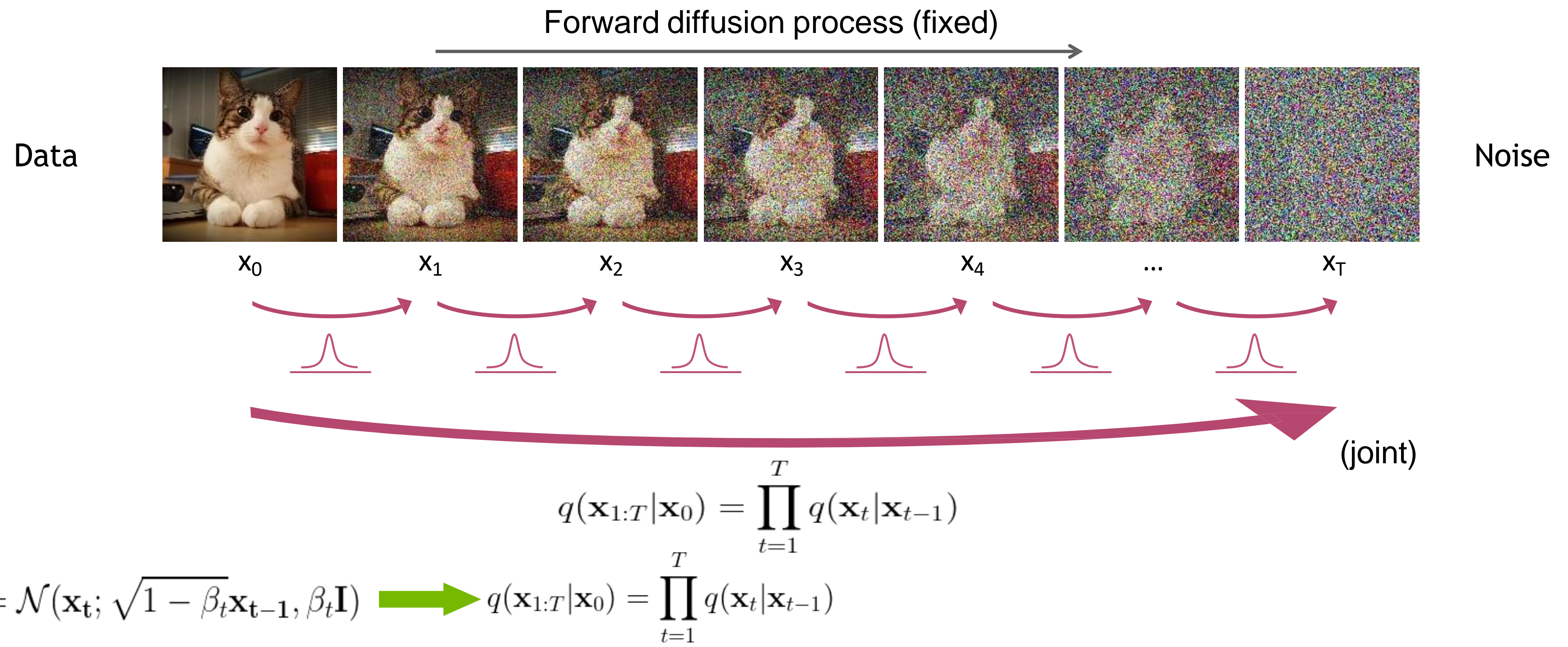
The Forward Diffusion Process

The formal definition of the forward process in T steps:



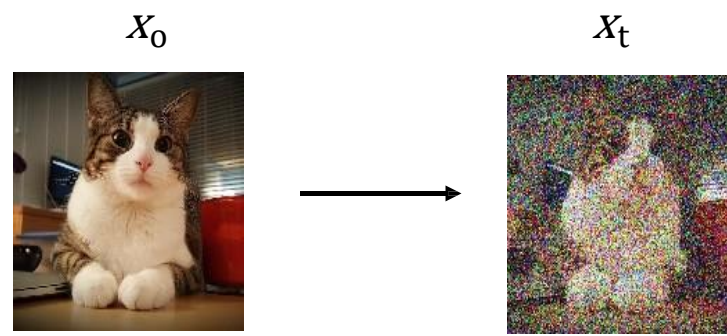
The Forward Diffusion Process

The formal definition of the forward process in T steps:



How do we do this in practice?

Sample image from the dataset,
generate noisy image using
forward process



$$q(\mathbf{x}_t \mid \mathbf{x}_0) = \mathcal{N}\left(\sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1 - \bar{\alpha}_t)\mathbf{I}\right)$$


Implementation of the Forward Process

$$q(\mathbf{x}_t \mid \mathbf{x}_0) = \mathcal{N}\left(\sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1 - \bar{\alpha}_t)\mathbf{I}\right) \quad \begin{aligned} \alpha_t &= 1 - \beta_t \\ \bar{\alpha}_t &= \prod_{i=1}^t \alpha_i \end{aligned}$$

1. Sample an image \mathbf{x}_0 from the dataset:



2. Sample noise $\epsilon \sim \mathcal{N}(0, \mathbf{I})$ (from a **standard** normal distribution)

3. Add $\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon \longrightarrow$ 

where

$$\begin{aligned} \alpha_t &= 1 - \beta_t \\ \bar{\alpha}_t &= \prod_{i=1}^t \alpha_i \end{aligned}$$


Implementation of the Forward Process

$$q(\mathbf{x}_t | \mathbf{x}_0) = \mathcal{N}\left(\sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1 - \bar{\alpha}_t)\mathbf{I}\right) \quad \alpha_t = 1 - \beta_t$$
$$\bar{\alpha}_t = \prod_{i=1}^t \alpha_i$$

1. Sample an image \mathbf{x}_0 from the dataset:



2. Sample noise $\mathbf{E} \sim \mathcal{N}(0, \mathbf{I})$ (from a **standard** normal distribution)

3. Add $\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t}\boldsymbol{\epsilon} \longrightarrow$ 

where

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

```
def noise_images(self, x, t):  
    sqrt_alpha_hat = torch.sqrt(self.alpha_hat[t][:, None, None, None])  
    sqrt_one_minus_alpha_hat = torch.sqrt(1 - self.alpha_hat[t][:, None, None, None])  
    E = torch.randn_like(x)  
    return sqrt_alpha_hat * x + sqrt_one_minus_alpha_hat * E, E
```

In this function we return the noisy image and the noise we used to add noise it to it. We both of these images for training.

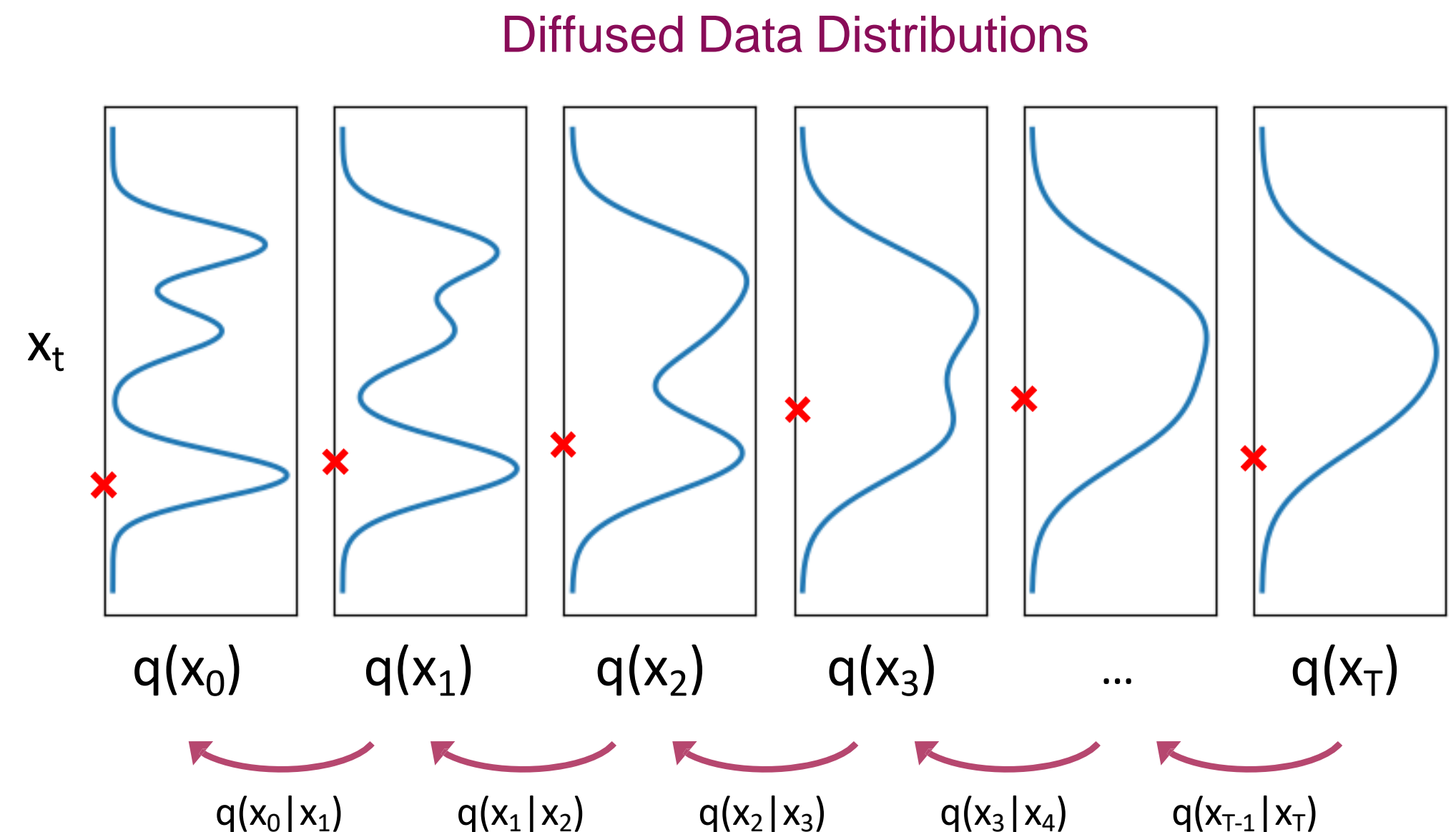
The reverse diffusion process: Generative Learning by Denoising

Recall, that the diffusion parameters are designed such that $q(\mathbf{x}_T) \approx \mathcal{N}(\mathbf{x}_T; \mathbf{0}, \mathbf{I})$

Generation:

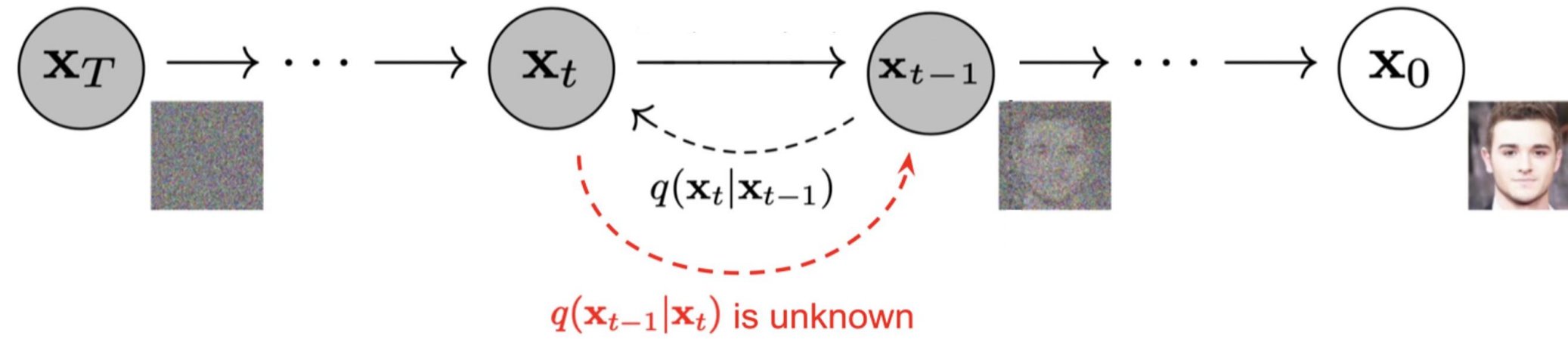
Sample $\mathbf{x}_T \sim \mathcal{N}(\mathbf{x}_T; \mathbf{0}, \mathbf{I})$

Iteratively sample $\mathbf{x}_{t-1} \sim \underbrace{q(\mathbf{x}_{t-1}|\mathbf{x}_t)}_{\text{True Denoising Dist.}}$

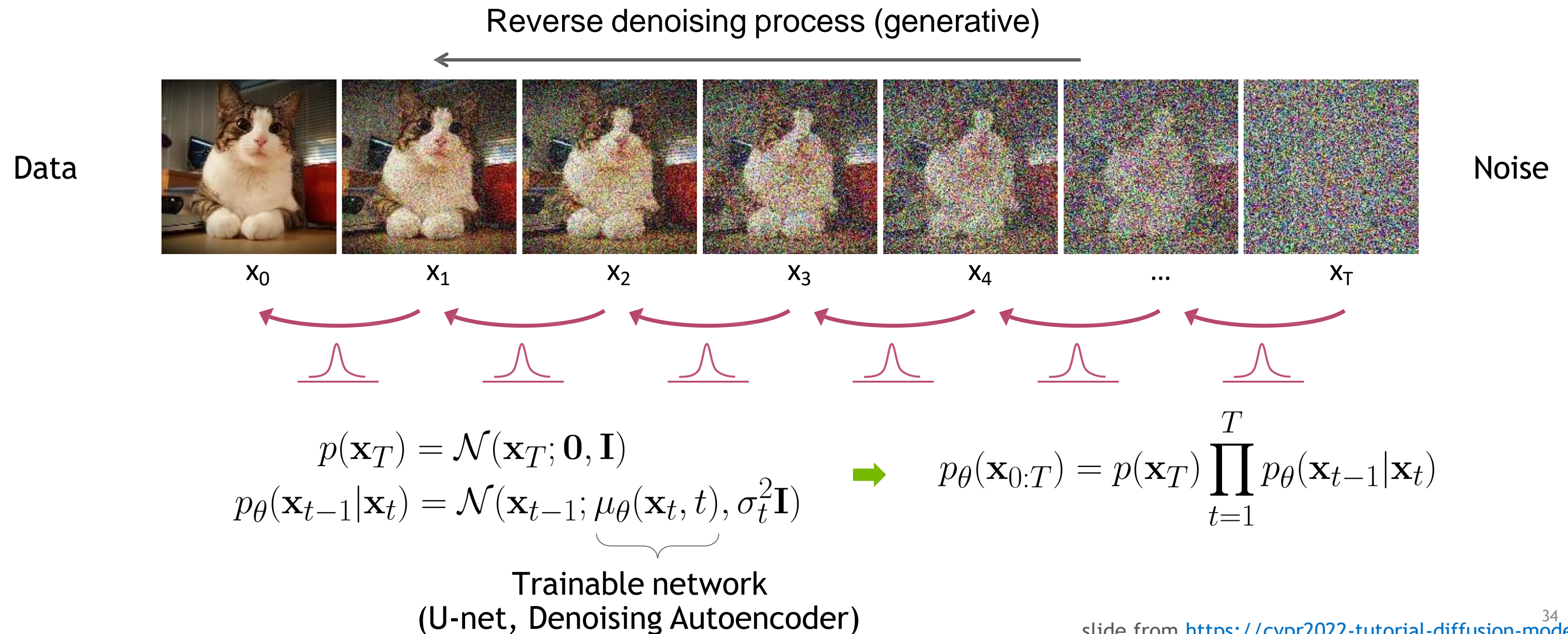


Can we approximate $q(\mathbf{x}_{t-1}|\mathbf{x}_t)$? Yes, we can use a Gaussian distribution if β_t is small in each forward diffusion step.

The reverse diffusion process: Generative Learning by Denoising

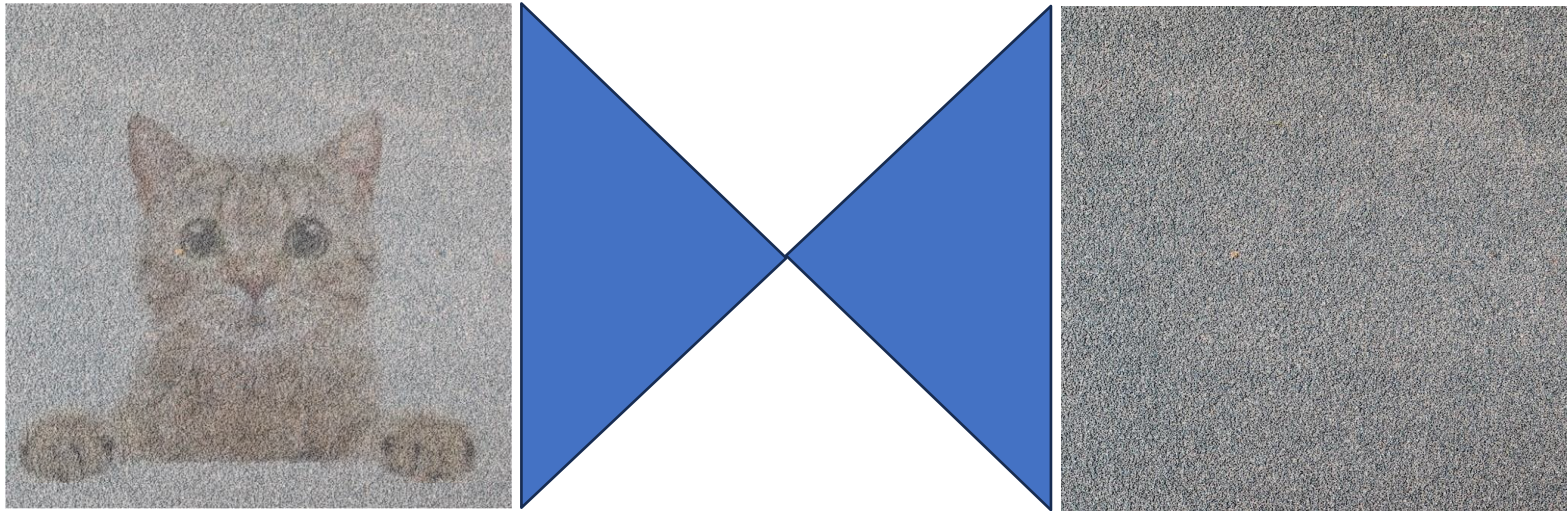


The reverse diffusion process: Generative Learning by Denoising



Neural Network that predicts noise

U-net



Input

Output

$$\nabla_{\theta} \left\| \epsilon - \epsilon_{\theta} \left(\sqrt{\bar{\alpha}_t} \mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t} \epsilon, t \right) \right\|^2$$

Summary

Algorithm 1 Training

```
1: repeat
2:    $\mathbf{x}_0 \sim q(\mathbf{x}_0)$ 
3:    $t \sim \text{Uniform}(\{1, \dots, T\})$ 
4:    $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ 
5:   Take gradient descent step on
        $\nabla_{\theta} \|\epsilon - \epsilon_{\theta}(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon, t)\|^2$ 
6: until converged
```

Algorithm 2 Sampling

```
1:  $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ 
2: for  $t = T, \dots, 1$  do
3:    $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ 
4:    $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left( \mathbf{x}_t - \frac{1 - \alpha_t}{\sqrt{1 - \bar{\alpha}_t}} \epsilon_{\theta}(\mathbf{x}_t, t) \right) + \sigma_t \mathbf{z}$ 
5: end for
6: return  $\mathbf{x}_0$ 
```

Summary

Algorithm 1 Training

```
1: repeat
2:    $\mathbf{x}_0 \sim q(\mathbf{x}_0)$ 
3:    $t \sim \text{Uniform}(\{1, \dots, T\})$ 
4:    $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ 
5:   Take gradient descent step on
        $\nabla_{\theta} \|\epsilon - \epsilon_{\theta}(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon, t)\|^2$ 
6: until converged
```

```
model = UNet().to(device)
optimizer = optim.AdamW(model.parameters(), lr=args.lr)
mse = nn.MSELoss()
diffusion = Diffusion(img_size=args.image_size, device=device)
logger = SummaryWriter(os.path.join("runs", args.run_name))
l = len(dataloader)

for epoch in range(args.epochs):
    logging.info(f"Starting epoch {epoch}:")
    pbar = tqdm(dataloader)
    for i, (images, _) in enumerate(pbar):
        images = images.to(device)
        t = diffusion.sample_timesteps(images.shape[0]).to(device)
        x_t, noise = diffusion.noise_images(images, t)
        predicted_noise = model(x_t, t)
        loss = mse(noise, predicted_noise)

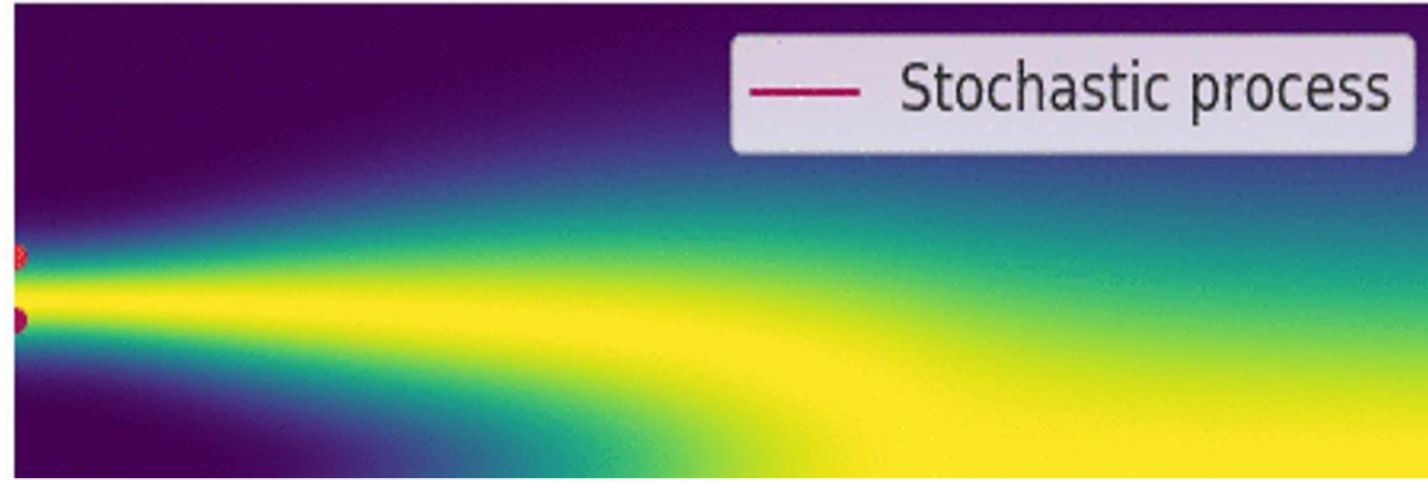
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Summary

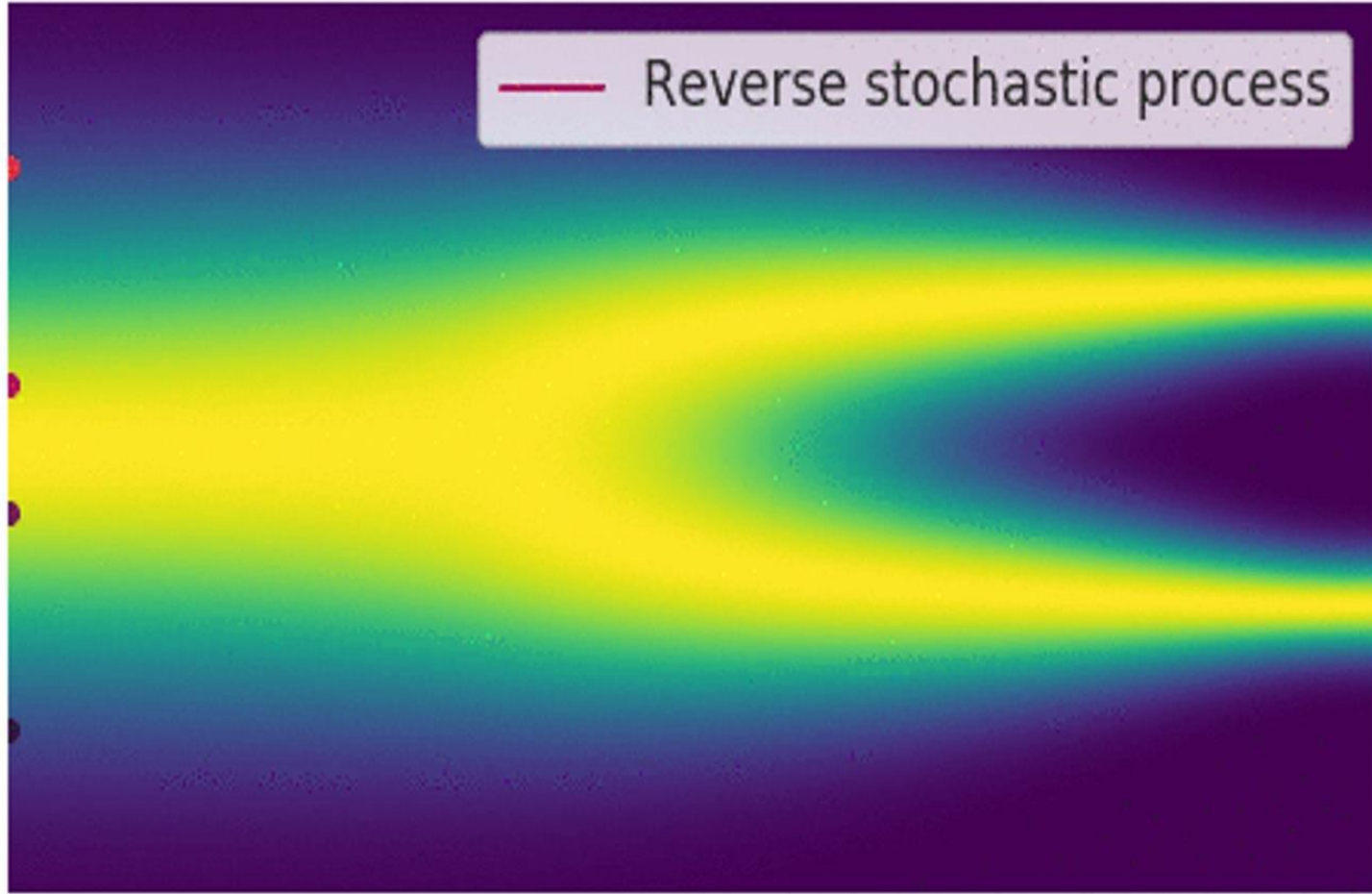
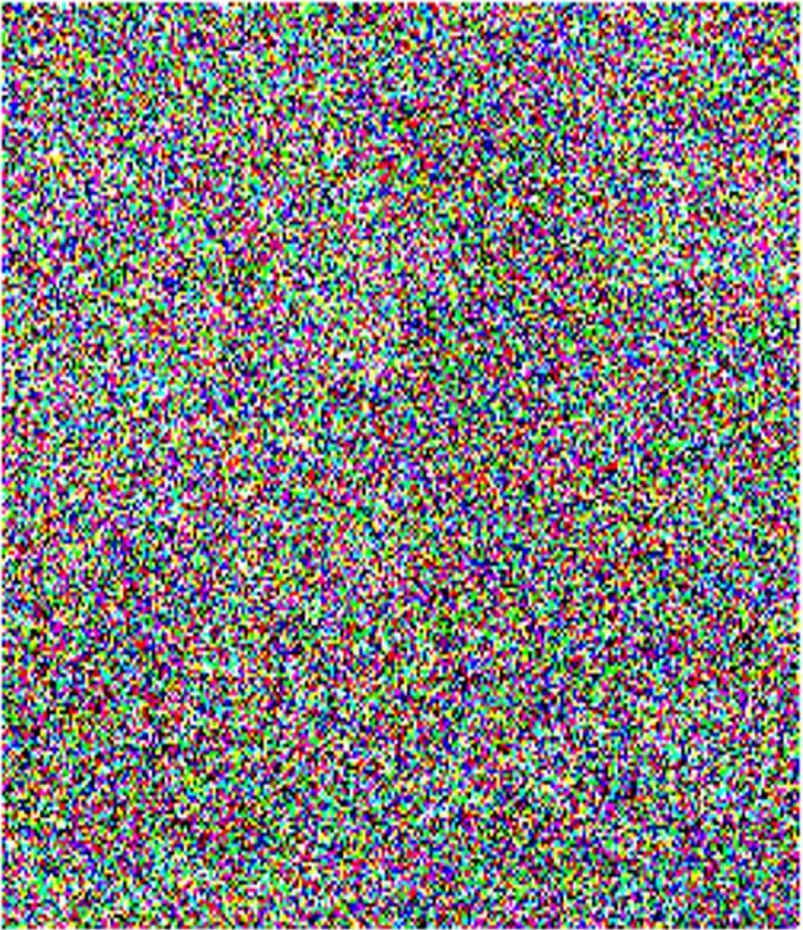
Algorithm 2 Sampling

```
1:  $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$   
2: for  $t = T, \dots, 1$  do  
3:    $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$   
4:    $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left( \mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\alpha_t}} \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t) \right) + \sigma_t \mathbf{z}$   
5: end for  
6: return  $\mathbf{x}_0$ 
```

```
def sample(self, model, n):  
    logging.info(f"Sampling {n} new images....")  
    model.eval()  
    with torch.no_grad():  
        x = torch.randn((n, 3, self.img_size, self.img_size)).to(self.device)  
        for i in tqdm(reversed(range(1, self.noise_steps)), position=0):  
            t = (torch.ones(n) * i).long().to(self.device)  
            predicted_noise = model(x, t)  
            alpha = self.alpha[t][:, None, None, None]  
            alpha_hat = self.alpha_hat[t][:, None, None, None]  
            beta = self.beta[t][:, None, None, None]  
            if i > 1:  
                noise = torch.randn_like(x)  
            else:  
                noise = torch.zeros_like(x)  
            x = 1 / torch.sqrt(alpha) * (x - ((1 - alpha) / (torch.sqrt(1 - alpha_hat))) * predicted_noise) + torch.sqrt(beta) * noise
```

Forward process: converting the image distribution to pure noise



Reverse process: sampling from the image distribution, starting with pure noise

Hugging face : diffusers

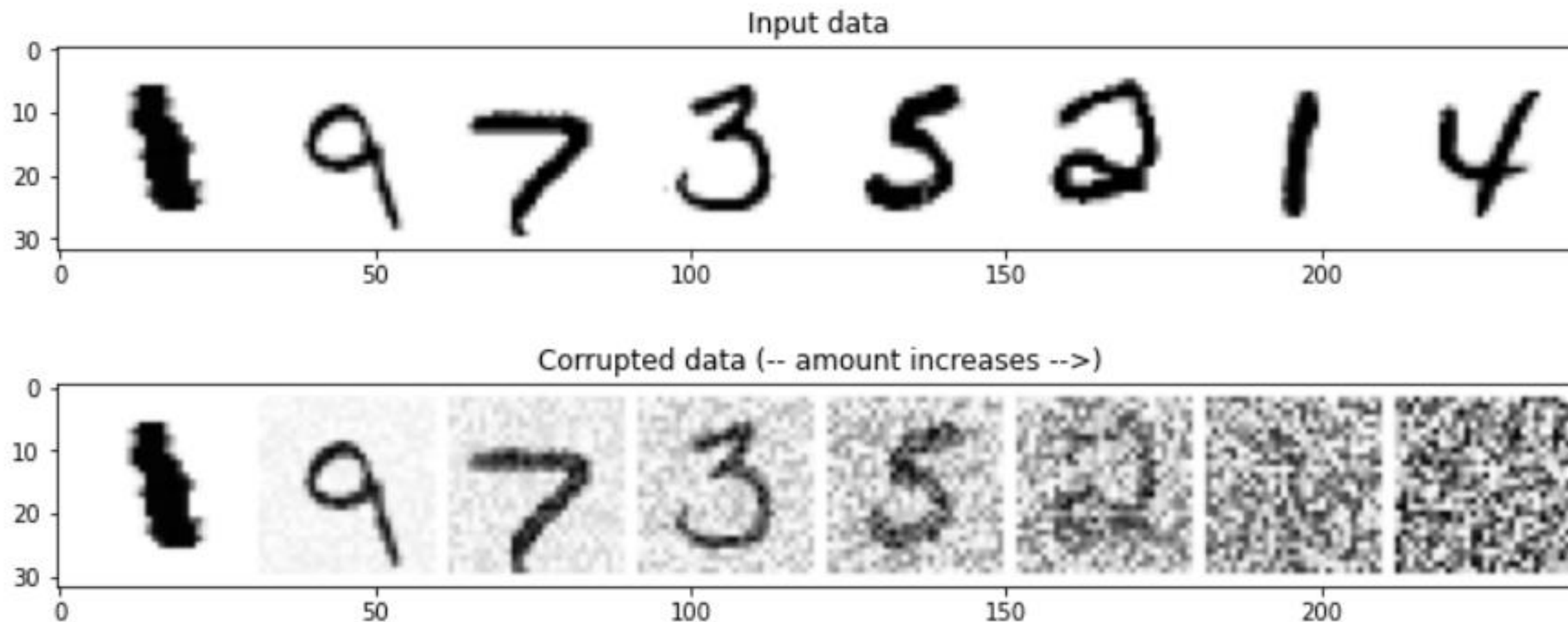
```
def corrupt(x, amount):  
    """Corrupt the input `x` by mixing it with noise according to `amount`"""  
    noise = torch.rand_like(x)  
    amount = amount.view(-1, 1, 1, 1) # Sort shape so broadcasting works  
    return x*(1-amount) + noise*amount
```

Hugging face : diffusers

```
# Plotting the input data
fig, axs = plt.subplots(2, 1, figsize=(12, 5))
axs[0].set_title('Input data')
axs[0].imshow(torchvision.utils.make_grid(x)[0], cmap='Greys')

# Adding noise
amount = torch.linspace(0, 1, x.shape[0]) # Left to right -> more corruption
noised_x = corrupt(x, amount)

# Plotting the noised version
axs[1].set_title('Corrupted data (-- amount increases -->)')
axs[1].imshow(torchvision.utils.make_grid(noised_x)[0], cmap='Greys');
```



Hugging face : diffusers

Training

```
# The training loop
for epoch in range(n_epochs):

    for x, y in train_dataloader:

        # Get some data and prepare the corrupted version
        x = x.to(device) # Data on the GPU
        noise_amount = torch.rand(x.shape[0]).to(device) # Pick random noise amounts
        noisy_x = corrupt(x, noise_amount) # Create our noisy x

        # Get the model prediction
        pred = net(noisy_x)

        # Calculate the loss
        loss = loss_fn(pred, x) # How close is the output to the true 'clean' x?

        # Backprop and update the params:
        opt.zero_grad()
        loss.backward()
        opt.step()

        # Store the loss for later
        losses.append(loss.item())
```

Hugging face : diffusers

Generation

```
#@markdown Sampling strategy: Break the process into 5 steps and move 1/5'th of the way there each time:
n_steps = 5
x = torch.rand(8, 1, 28, 28).to(device) # Start from random
step_history = [x.detach().cpu()]
pred_output_history = []

for i in range(n_steps):
    with torch.no_grad(): # No need to track gradients during inference
        pred = net(x) # Predict the denoised x0
        pred_output_history.append(pred.detach().cpu()) # Store model output for plotting
        mix_factor = 1/(n_steps - i) # How much we move towards the prediction
        x = x*(1-mix_factor) + pred*mix_factor # Move part of the way there
        step_history.append(x.detach().cpu()) # Store step for plotting

fig, axs = plt.subplots(n_steps, 2, figsize=(9, 4), sharex=True)
axs[0,0].set_title('x (model input)')
axs[0,1].set_title('model prediction')
for i in range(n_steps):
    axs[i, 0].imshow(torchvision.utils.make_grid(step_history[i])[0].clip(0, 1), cmap='Greys')
    axs[i, 1].imshow(torchvision.utils.make_grid(pred_output_history[i])[0].clip(0, 1), cmap='Greys')
```

