MP5: Training Your Diffusion Model!

Setup environment

```
# Import essential modules. Feel free to add whatever you need.
import matplotlib.pyplot as plt
import torch # added
from torch import optim # added
from torch import nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision.datasets import MNIST
from torchvision.transforms import ToTensor
```

Visualization helper function

```
def visualize_images_with_titles(images: torch.Tensor, column_names: list[str]):
    """
    Visualize images as a grid and title the columns with the provided names.

    Args:
        images: (N, C, H, W) tensor of images, where N is (number of rows * number of columns)
        column_names: List of column names for the titles.

    Example usage:
    visualize_images_with_titles(torch.randn(16, 1, 32, 32), ['1', '2', '3', '4'])
    """
    num_images, num_columns = images.shape[0], len(column_names)
    assert num_images % num_columns == 0, 'Number of images must be a multiple of the number of columns.'

    num_rows = num_images // num_columns
    fig, axes = plt.subplots(num_rows, num_columns, figsize=(num_columns * 1, num_rows * 1))

    for i, ax in enumerate(axes.flat):
        img = images[i].permute(1, 2, 0).cpu().numpy()
        ax.imshow(img, cmap='gray')
        ax.axis('off')
        if i < num_columns:
            ax.set_title(column_names[i % num_columns])
```

```
    plt.tight_layout()
    plt.show()
```

Part 1: Training a Single-step Denoising UNet

Implementing Simple and Composed Ops

```
class Conv(nn.Module):
    def __init__(self, in_channels: int, out_channels: int):
        super().__init__()
        self.conv = nn.Conv2d(in_channels, out_channels,
kernel_size=3, stride=1, padding=1)
        self.bn   = nn.BatchNorm2d(out_channels)
        self.gelu = nn.GELU()

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.conv(x)
        x = self.bn(x)
        x = self.gelu(x)
        return x
        raise NotImplementedError()


class DownConv(nn.Module):
    def __init__(self, in_channels: int, out_channels: int):
        super().__init__()
        self.down_conv = nn.Conv2d(in_channels, out_channels,
kernel_size=3, stride=2, padding=1)
        self.bn   = nn.BatchNorm2d(out_channels)
        self.gelu = nn.GELU()

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.down_conv(x)
        x = self.bn(x)
        x = self.gelu(x)
        return x
        raise NotImplementedError()


class UpConv(nn.Module):
    def __init__(self, in_channels: int, out_channels: int):
        super().__init__()
        self.up_conv = nn.ConvTranspose2d(in_channels, out_channels,
kernel_size=4, stride=2, padding=1)
        self.bn      = nn.BatchNorm2d(out_channels)
        self.gelu    = nn.GELU()
```

```python
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.up_conv(x)
        x = self.bn(x)
        x = self.gelu(x)
        return x
        raise NotImplementedError()


class Flatten(nn.Module):
    def __init__(self):
        super().__init__()
        self.avg_pool = nn.AvgPool2d(kernel_size=7)
        self.gelu     = nn.GELU()

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.avg_pool(x)
        x = self.gelu(x)
        return x
        raise NotImplementedError()


class Unflatten(nn.Module):
    def __init__(self, in_channels: int):
        super().__init__()
        self.unflatten = nn.ConvTranspose2d(in_channels, in_channels,
kernel_size=7, stride=7, padding=0)
        self.bn       = nn.BatchNorm2d(in_channels)
        self.gelu     = nn.GELU()

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.unflatten(x)
        x = self.bn(x)
        x = self.gelu(x)
        return x
        raise NotImplementedError()


class ConvBlock(nn.Module):
    def __init__(self, in_channels: int, out_channels: int):
        super().__init__()
        self.conv1 = Conv(in_channels, out_channels)
        self.conv2 = Conv(out_channels, out_channels)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.conv1(x)
        x = self.conv2(x)
        return x
        raise NotImplementedError()
```

```python
class DownBlock(nn.Module):
    def __init__(self, in_channels: int, out_channels: int):
        super().__init__()
        self.down_conv = DownConv(in_channels, out_channels)
        self.conv      = ConvBlock(out_channels, out_channels)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.down_conv(x)
        x = self.conv(x)
        return x
        raise NotImplementedError()


class UpBlock(nn.Module):
    def __init__(self, in_channels: int, out_channels: int):
        super().__init__()
        self.up_conv = UpConv(in_channels, out_channels)
        self.conv    = ConvBlock(out_channels, out_channels)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.up_conv(x)
        x = self.conv(x)
        return x
        raise NotImplementedError()
```

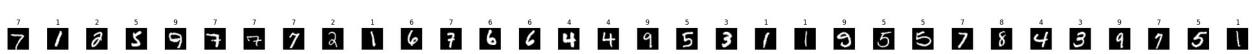
Implementing Unconditional UNet

```python
class UnconditionalUNet(nn.Module):
    def __init__(
        self,
        in_channels: int,
        num_hiddens: int,
    ):
        super().__init__()
        self.convblock1 = ConvBlock(in_channels, num_hiddens)
        self.convblock2 = ConvBlock(2*num_hiddens, num_hiddens)
        self.downblock1 = DownBlock(num_hiddens, num_hiddens)
        self.downblock2 = DownBlock(num_hiddens, 2*num_hiddens)
        self.flatten    = Flatten()
        self.unflatten  = Unflatten(2*num_hiddens)
        self.upblock1   = UpBlock(4*num_hiddens, num_hiddens)
        self.upblock2   = UpBlock(2*num_hiddens, num_hiddens)
        self.conv       = nn.Conv2d(num_hiddens, 1, kernel_size=3,
stride=1, padding=1)
        self.convblock1 = ConvBlock(in_channels, num_hiddens)
        self.convblock2 = ConvBlock(2*num_hiddens, num_hiddens)
        self.downblock1 = DownBlock(num_hiddens, num_hiddens)
        self.downblock2 = DownBlock(num_hiddens, 2*num_hiddens)
```

```
        self.flatten    = Flatten()
        self.unflatten  = Unflatten(2*num_hiddens)
        self.upblock1   = UpBlock(4*num_hiddens, num_hiddens)
        self.upblock2   = UpBlock(2*num_hiddens, num_hiddens)
        self.conv       = nn.Conv2d(num_hiddens, 1, kernel_size=3,
stride=1, padding=1)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        assert x.shape[-2:] == (28, 28), "Expect input shape to be
(28, 28)."
        x = self.convblock1(x)
        x1 = x
        x = self.downblock1(x)
        x2 = x
        x = self.downblock2(x)
        x3 = x
        x = self.flatten(x)
        x = self.unflatten(x)
        x = torch.cat((x,x3),dim=1)
        x = self.upblock1(x)
        x = torch.cat((x,x2),dim=1)
        x = self.upblock2(x)
        x = torch.cat((x,x1),dim=1)
        x = self.convblock2(x)
        x = self.conv(x)
        return x
        raise NotImplementedError()
```

Visualizing the noising process

```
dataset = MNIST(root="data", download=True, transform=ToTensor(),
train=True)

dataloader = DataLoader(dataset, batch_size=32, shuffle=True)
images, labels = next(iter(dataloader))
visualize_images_with_titles(images, [str(label.item()) for label in
labels])
```



Training a Single-Step Unconditional UNet

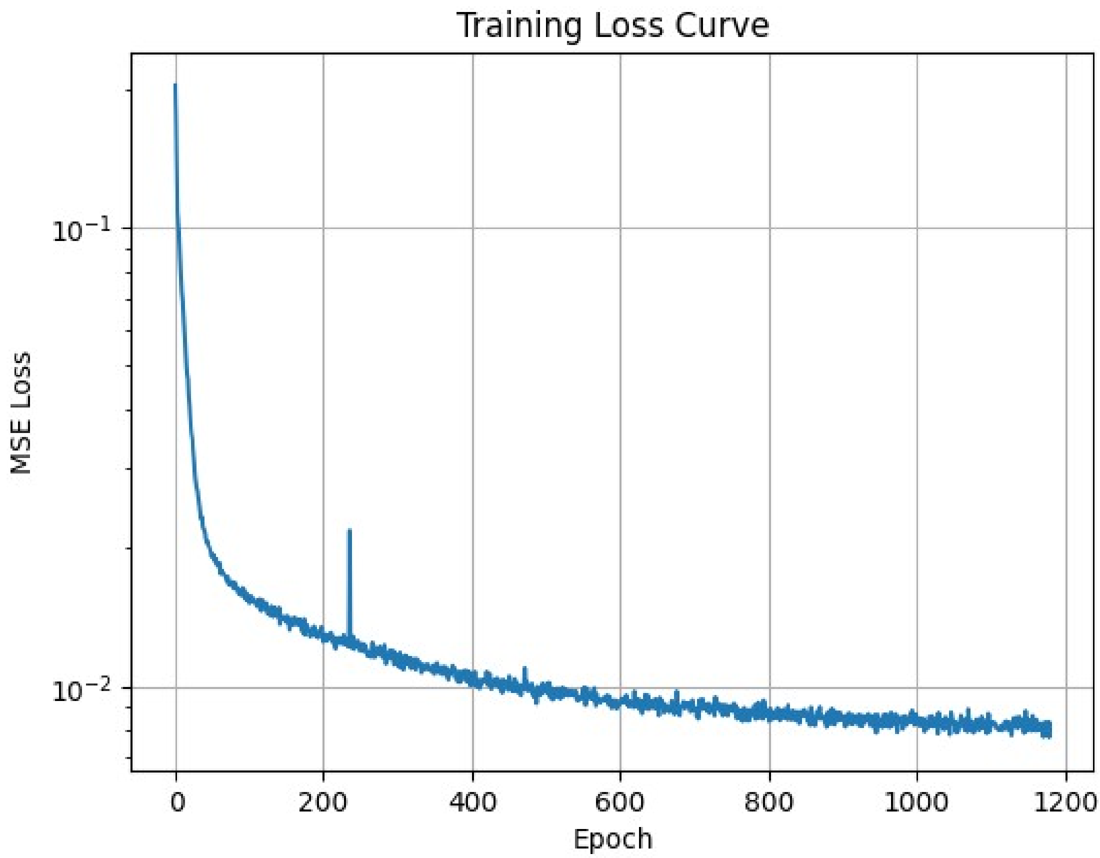
- Plot the loss curve
- Sample results on the test set

```
import torchvision.utils as vutils

dataset = MNIST(root='data', download=True, transform=ToTensor(),
train=True)
dataloader = DataLoader(dataset, batch_size=256, shuffle=True)
```

```python
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model = UnconditionalUNet(in_channels=1, num_hiddens=128).to(device)
loss_fn = nn.MSELoss()

num_epochs = 5

test_dataset = MNIST(root='data', download=True, transform=ToTensor(),
train=False)
test_loader = DataLoader(test_dataset, batch_size=5, shuffle=True)

optimizer = optim.Adam(model.parameters(), lr=1e-4)
avg_loss_history = []
loss_history = []

for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for batch in dataloader:
        images, _ = batch
        images = images.to(device)

        noisy_images = images + 0.5 * torch.randn_like(images)

        outputs = model(noisy_images)
        loss = loss_fn(outputs, images)

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

        loss_history.append(loss.item())
        running_loss += loss.item()

    avg_loss = running_loss / len(dataloader)
    loss_history.append(avg_loss)
    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {avg_loss:.4f}")

plt.plot(loss_history)
plt.title("Training Loss Curve")
plt.xlabel("Epoch")
plt.yscale('log')
plt.ylabel("MSE Loss")
plt.grid()
plt.show()
```

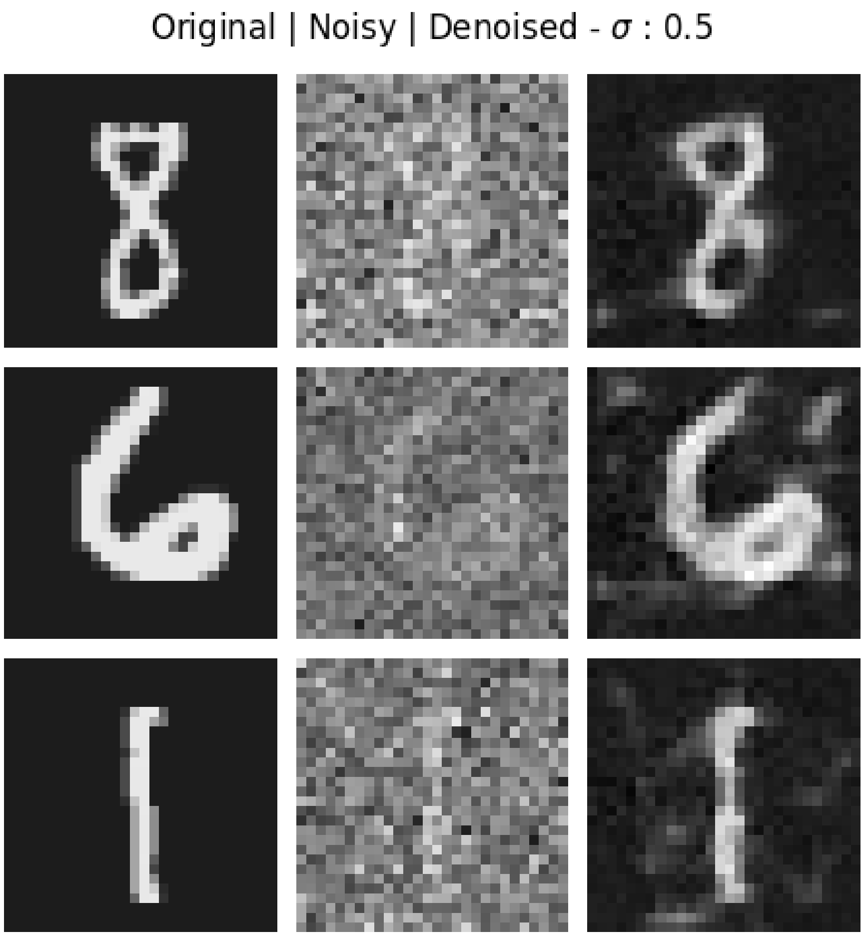
```
Epoch [1/5], Loss: 0.0219
Epoch [2/5], Loss: 0.0110
Epoch [3/5], Loss: 0.0094
```

```
Epoch [4/5], Loss: 0.0087
Epoch [5/5], Loss: 0.0083
```



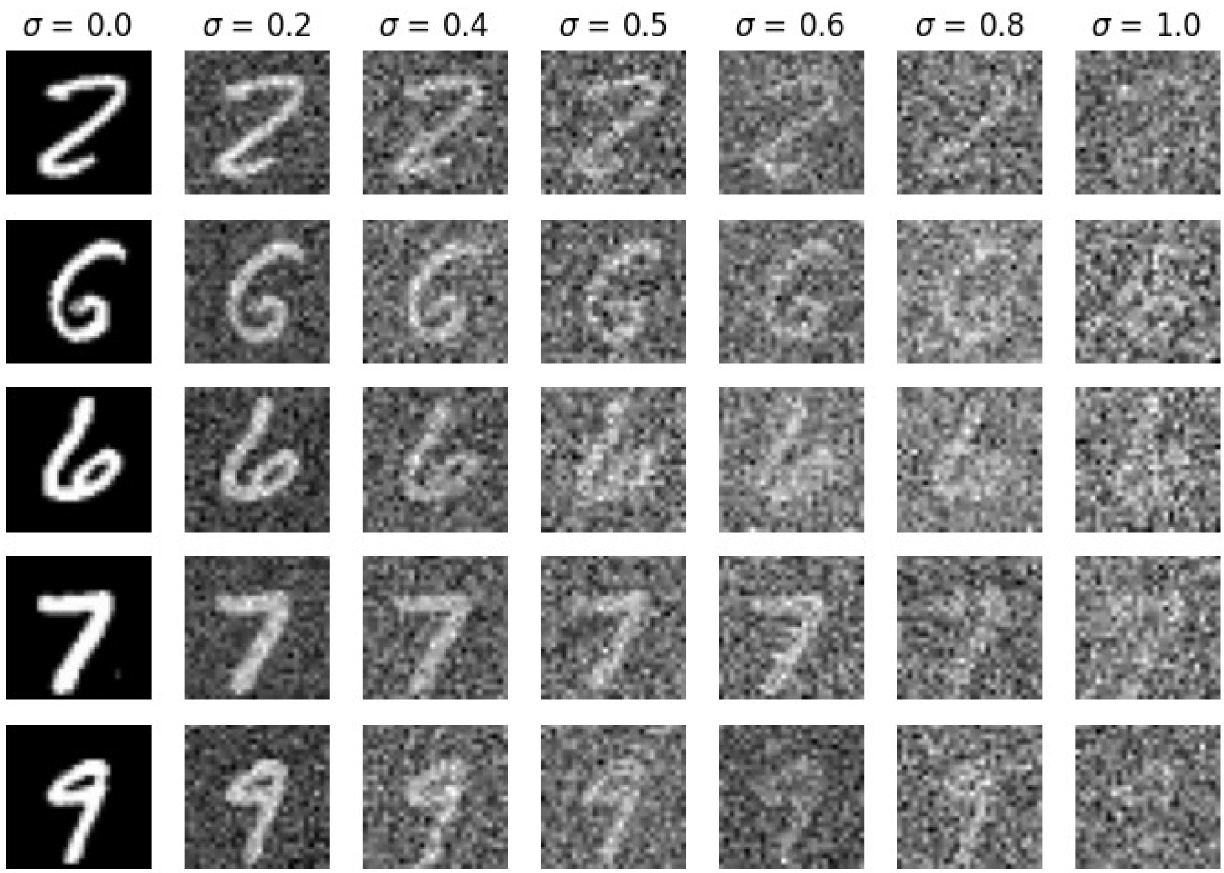

```
model.eval()
# test_images, _ = next(iter(test_loader))
test_images = test_images.to(device)

noisy_test_images = test_images + 0.7*torch.randn_like(test_images)

with torch.no_grad():
    denoised_images = model(noisy_test_images)
# Visualize: original noisy and denoised images

def show_image_triplets_columnwise(clean_imgs, noisy_imgs,
denoised_imgs, num_images=3, sigma=0.5):
    """
    Show triplets (Clean, Noisy, Denoised) columnwise
    """
    clean_imgs = clean_imgs[:num_images]
    noisy_imgs = noisy_imgs[:num_images]
    noisy_imgs = (noisy_imgs - noisy_imgs.min(dim=2, keepdim=True)
[0].min(dim=3, keepdim=True)[0])
```

```
    noisy_imgs = noisy_imgs / (noisy_imgs.max(dim=2, keepdim=True)
[0].max(dim=3, keepdim=True)[0] + 1e-8)

    # Stack clean, noisy, denoised
    denoised_imgs = denoised_imgs[:num_images]

    # Stack them: (num_images, 3, C, H, W)
    triplets = torch.stack([clean_imgs, noisy_imgs, denoised_imgs],
dim=1)  # (N, 3, C, H, W)

    # Now reshape to (3*num_images, C, H, W) for columnwise display
    triplets = triplets.view(-1, *clean_imgs.shape[1:])  # (3*N, C, H,
W)

    # Make a grid
    grid = vutils.make_grid(triplets, nrow=3, pad_value=1.0,
padding=2, normalize=True)

    plt.figure(figsize=(9, 2*num_images))
    img = grid.cpu().permute(1, 2, 0)
    plt.imshow(img)
    plt.title(f'Original | Noisy | Denoised - $\sigma$ : {sigma}')
    plt.axis('off')
    plt.show()

show_image_triplets_columnwise(test_images, noisy_test_images,
denoised_images)
```



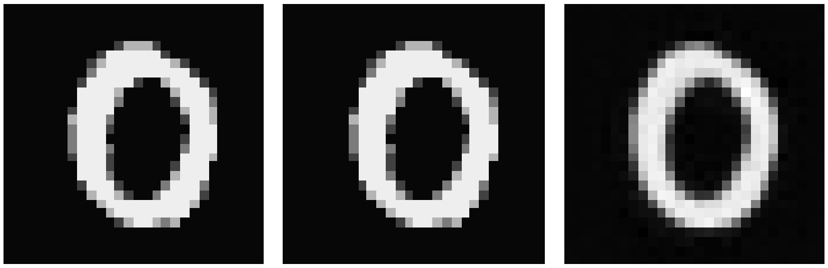
Out-of-Distribution Testing

```
model.eval()
test_images, _ = next(iter(test_loader))
sigma_list = [0.0, 0.2, 0.4, 0.5, 0.6, 0.8, 1.0]
sampled_images_list = []
titles = []
test_images = test_images.to(device)

for i in sigma_list:
    sampled =  test_images + i*torch.randn_like(test_images)
    sampled_images_list.append(sampled)
    titles.append(f'$\sigma$ = {i}')

all_samples = torch.hstack(sampled_images_list)

axes = visualize_images_with_titles(all_samples, titles)
```

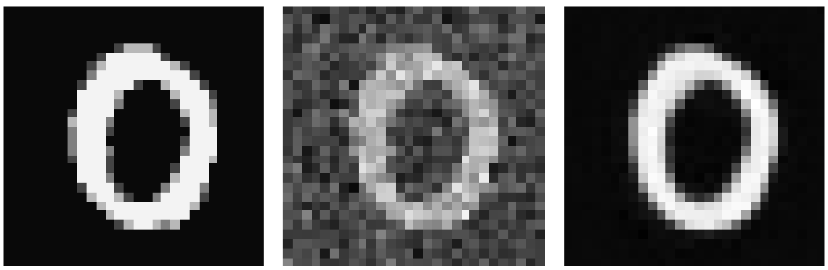



```
model.eval()
test_images, _ = next(iter(test_loader))

for sigma in [0.0, 0.2, 0.4, 0.5, 0.6, 0.8, 1.0] :
    test_images = test_images.to(device)
    noisy_test_images = test_images +
sigma*torch.randn_like(test_images)

    with torch.no_grad():
        denoised_images = model(noisy_test_images)

    show_image_triplets_columnwise(test_images, noisy_test_images,
denoised_images, num_images=1, sigma=sigma)
```

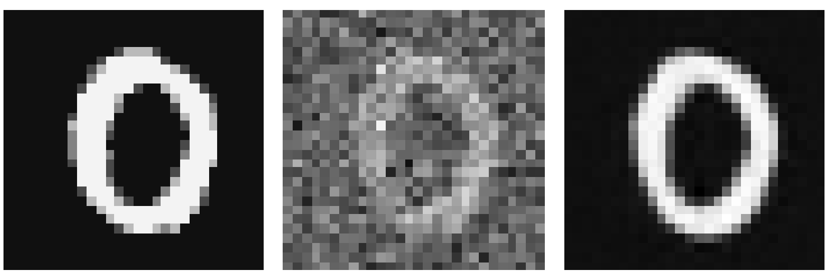
Original | Noisy | Denoised - σ : 0.0



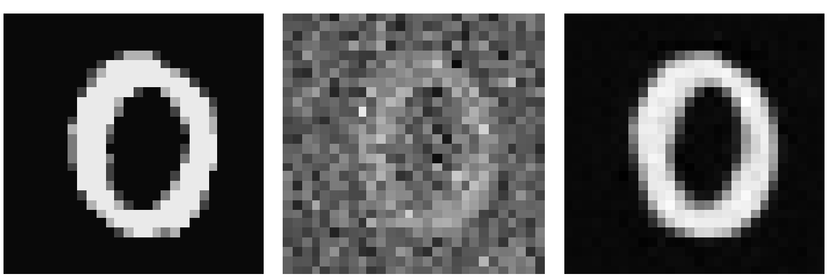
Original | Noisy | Denoised - σ : 0.2

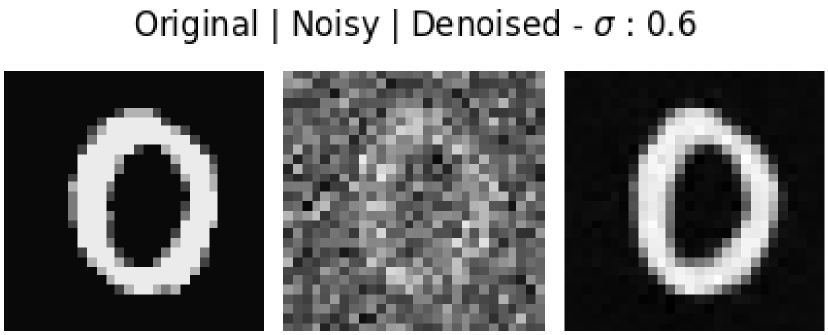


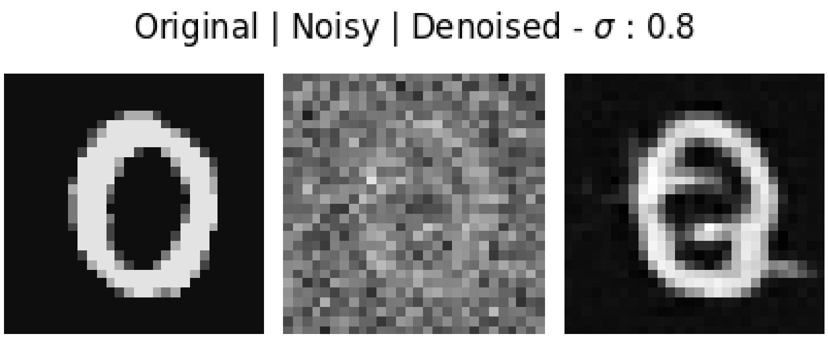
Original | Noisy | Denoised - σ : 0.4

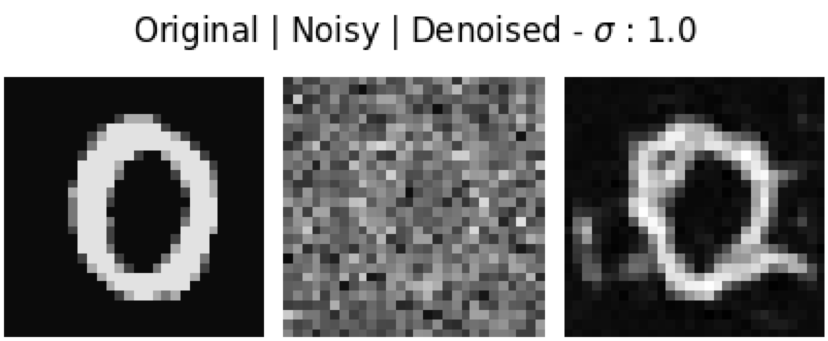


Original | Noisy | Denoised - σ : 0.5









Part 2: Training a Diffusion Model

Implementing a Time-conditioned UNet

```
class FCBlock(nn.Module):
    def __init__(self, in_channels: int, out_channels: int):
        super().__init__()
        self.fc1 = nn.Linear(in_channels, out_channels)
        self.gelu = nn.GELU()
        self.fc2 = nn.Linear(out_channels, out_channels)
```

```python
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.fc1(x)
        x = self.gelu(x)
        x = self.fc2(x)
        return x.unsqueeze(-1).unsqueeze(-1)
        raise NotImplementedError()


class TimeConditionalUNet(nn.Module):
    def __init__(
        self,
        in_channels: int,
        num_classes: int,
        num_hiddens: int,
    ):
        super().__init__()
        self.convblock1 = ConvBlock(in_channels, num_hiddens)
        self.convblock2 = ConvBlock(2*num_hiddens, num_hiddens)
        self.downblock1 = DownBlock(num_hiddens, num_hiddens)
        self.downblock2 = DownBlock(num_hiddens, 2*num_hiddens)
        self.flatten    = Flatten()
        self.unflatten  = Unflatten(2*num_hiddens)
        self.upblock1   = UpBlock(4*num_hiddens, num_hiddens)
        self.upblock2   = UpBlock(2*num_hiddens, num_hiddens)
        self.conv       = nn.Conv2d(num_hiddens, 1, kernel_size=3,
stride=1, padding=1)
        self.convblock1 = ConvBlock(in_channels, num_hiddens)
        self.convblock2 = ConvBlock(2*num_hiddens, num_hiddens)
        self.downblock1 = DownBlock(num_hiddens, num_hiddens)
        self.downblock2 = DownBlock(num_hiddens, 2*num_hiddens)
        self.flatten    = Flatten()
        self.unflatten  = Unflatten(2*num_hiddens)
        self.upblock1   = UpBlock(4*num_hiddens, num_hiddens)
        self.upblock2   = UpBlock(2*num_hiddens, num_hiddens)
        self.conv       = nn.Conv2d(num_hiddens, 1, kernel_size=3,
stride=1, padding=1)
        self.fcblock1   = FCBlock(1, 2*num_hiddens)
        self.fcblock2   = FCBlock(1, num_hiddens)

    def forward(
        self,
        x: torch.Tensor,
        t: torch.Tensor,
    ) -> torch.Tensor:
        """
        Args:
            x: (N, C, H, W) input tensor.
            t: (N,) normalized time tensor.
```

```
        Returns:
            (N, C, H, W) output tensor.
        """
        assert x.shape[-2:] == (28, 28), "Expect input shape to be
(28, 28)."
        x = self.convblock1(x)
        x1 = x
        x = self.downblock1(x)
        x2 = x
        x = self.downblock2(x)
        x3 = x
        x = self.flatten(x)
        x = self.unflatten(x) + self.fcblock1(t)
        x = torch.cat((x,x3),dim=1)
        x = self.upblock1(x) + self.fcblock2(t)
        x = torch.cat((x,x2),dim=1)
        x = self.upblock2(x)
        x = torch.cat((x,x1),dim=1)
        x = self.convblock2(x)
        x = self.conv(x)
        return x
        raise NotImplementedError()
```

Implementing DDPM Forward and Inverse Process for Time-conditioned Denoising

```
def ddpm_schedule(beta1: float, beta2: float, num_ts: int) -> dict:
    """Constants for DDPM training and sampling.

    Arguments:
        beta1: float, starting beta value.
        beta2: float, ending beta value.
        num_ts: int, number of timesteps.

    Returns:
        dict with keys:
            betas: linear schedule of betas from beta1 to beta2.
            alphas: 1 - betas.
            alpha_bars: cumulative product of alphas.
    """
    assert beta1 < beta2 < 1.0, "Expect beta1 < beta2 < 1.0."
    betas = torch.linspace(beta1, beta2, num_ts+1, device='cuda')
    alphas = 1 - betas
    alpha_bars = torch.cumprod(alphas, dim=0)

    return {
        "betas"     : betas,
        "alphas"    : alphas,
        "alpha_bars": alpha_bars
```

```
    }
    raise NotImplementedError()
```

Forward Pass :

```
def ddpm_forward(
    unet: TimeConditionalUNet,
    ddpm_schedule: dict,
    x_0: torch.Tensor,
    num_ts: int,
) -> torch.Tensor:
    """Algorithm 1 of the DDPM paper.

    Args:
        unet: TimeConditionalUNet
        ddpm_schedule: dict
        x_0: (N, C, H, W) input tensor.
        num_ts: int, number of timesteps.
    Returns:
        (,) diffusion loss.
    """
    unet.train()
    # YOUR CODE HERE.
    t = torch.randint(1,num_ts+1,size=(x_0.size(0),1),device="cuda")
    alpha_bar_t = ddpm_schedule["alpha_bars"][t.squeeze(-1)]
    alpha_bar_t = alpha_bar_t.view(-1,1,1,1)
    e = torch.randn_like(x_0)
    x_t = torch.sqrt(alpha_bar_t)*x_0 + torch.sqrt(1-alpha_bar_t)*e
    e_hat = unet(x_t, t.float()/num_ts)
    loss = nn.MSELoss()
    return loss(e,e_hat)
    raise NotImplementedError()
```

Sampling

```
@torch.inference_mode()
def ddpm_sample(
    unet: TimeConditionalUNet,
    ddpm_schedule: dict,
    img_wh: tuple[int, int],
    num_ts: int,
    seed: int = 0,
) -> torch.Tensor:
    """Algorithm 2 of the DDPM paper with classifier-free guidance.

    Args:
        unet: TimeConditionalUNet
        ddpm_schedule: dict
        img_wh: (H, W) output image width and height.
```

```
        num_ts: int, number of timesteps.
        seed: int, random seed.

    Returns:
        (N, C, H, W) final sample.
    """
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    unet.eval()
    # YOUR CODE HERE.
    betas = ddpm_schedule["betas"]
    alphas = ddpm_schedule["alphas"]
    alpha_bars = ddpm_schedule["alpha_bars"]
    N   = 5
    x_t = torch.randn(N,1,img_wh[0],img_wh[1],device='cuda')
    for t in range(num_ts,0,-1):
        alpha_t = alphas[t]
        beta_t  = betas[t]
        alpha_bar_t = alpha_bars[t]
        alpha_bar_t_1 = alpha_bars[t-1]
        if t>1 :
            z = torch.randn_like(x_t,device='cuda')
        else :
            z = torch.zeros_like(x_t)

        x_0_hat = (x_t - torch.sqrt(torch.abs(1-
alpha_bar_t))*unet(x_t,
t/num_ts*torch.ones((N,1),device='cuda')))/torch.sqrt(torch.abs(alpha_
bar_t))
        x_t =   ( torch.sqrt(torch.abs(alpha_bar_t_1))*beta_t*x_0_hat
+ torch.sqrt(torch.abs(alpha_t))*(1-alpha_bar_t_1)*x_t )/((1-
alpha_bar_t)+1e-5) + torch.sqrt(torch.abs(beta_t))*z

    return x_t
    raise NotImplementedError()

class DDPM(nn.Module):
    def __init__(
        self,
        unet: TimeConditionalUNet,
        betas: tuple[float, float] = (1e-4, 0.02),
        num_ts: int = 300,
        p_uncond: float = 0.1,
    ):
        super().__init__()
        self.unet = unet
        self.num_ts = num_ts
        self.p_uncond = p_uncond

        self.ddpm_schedule = ddpm_schedule(betas[0], betas[1], num_ts)
```

```python
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        Args:
            x: (N, C, H, W) input tensor.

        Returns:
            (,) diffusion loss.
        """
        return ddpm_forward(
            self.unet, self.ddpm_schedule, x, self.num_ts
        )

    @torch.inference_mode()
    def sample(
        self,
        img_wh: tuple[int, int],
        seed: int = 0,
    ):
        return ddpm_sample(
            self.unet, self.ddpm_schedule, img_wh, self.num_ts, seed
        )
```

Training the Time-conditioned UNet

- Plot the loss curve
- Sample results on the test set

```python
dataset = MNIST(root='data', download=True, transform=ToTensor(),
train=True)
dataloader = DataLoader(dataset, batch_size=128, shuffle=True)

device = 'cuda' if torch.cuda.is_available() else 'cpu'

unet = TimeConditionalUNet(in_channels=1, num_classes=1,
num_hiddens=64).to(device)
model = DDPM(unet=unet, betas=(1e-4, 0.02), num_ts=300).to(device)

epochs = 20
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer,
gamma=0.1**(1/epochs))

epoch_losses = []
losses = []

for epoch in range(epochs):
    model.train()
    total_loss = 0

    for batch_idx, (x, _) in enumerate(dataloader):
```

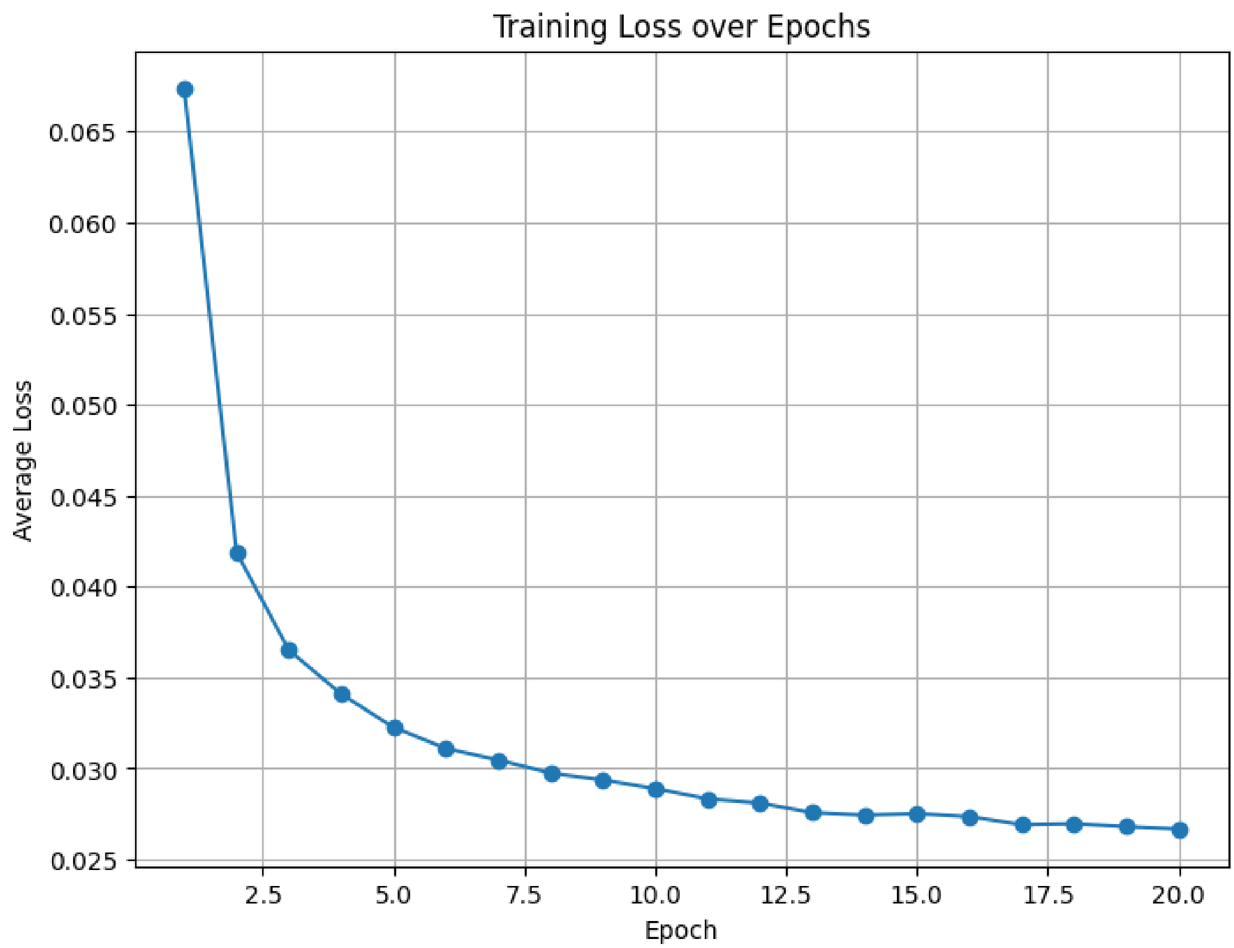
```
        x = x.to(device)

        optimizer.zero_grad()
        loss = model(x)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

        if batch_idx % 2 == 0 :
            losses.append(loss.item())

        if batch_idx % 100 == 0:
            print(f"Epoch [{epoch+1}/{epochs}] Batch [{batch_idx}]
Loss: {loss.item():.4f}")

    scheduler.step()

    avg_loss = total_loss / len(dataloader)
    epoch_losses.append(avg_loss)

    print(f"Epoch [{epoch+1}/{epochs}] Avg Loss: {avg_loss:.4f}")

    if (epoch + 1) % 5 == 0 or epoch+1 == 1:
        torch.save(model.state_dict(),
f"./models/model_epoch_{epoch+1}.pth")
        print(f"Model saved at epoch {epoch+1}")

print("Training complete!")

plt.figure(figsize=(8,6))
plt.plot(range(1, epochs+1), epoch_losses, marker='o')
plt.xlabel('Epoch')
plt.ylabel('Average Loss')
plt.title('Training Loss over Epochs')
plt.grid(True)
plt.show()

plt.figure(figsize=(8,6))
plt.plot(losses)
plt.xlabel('Step')
plt.yscale('log')
plt.ylabel('Loss')
plt.title('Training Loss over steps')
plt.grid(True)
plt.show()
```

```
Epoch [1/20] Batch [0] Loss: 1.1082
Epoch [1/20] Batch [100] Loss: 0.0733
Epoch [1/20] Batch [200] Loss: 0.0479
```

```
Epoch [1/20] Batch [300] Loss: 0.0500
Epoch [1/20] Batch [400] Loss: 0.0459
Epoch [1/20] Avg Loss: 0.0674
Model saved at epoch 1
Epoch [2/20] Batch [0] Loss: 0.0390
Epoch [2/20] Batch [100] Loss: 0.0355
Epoch [2/20] Batch [200] Loss: 0.0371
Epoch [2/20] Batch [300] Loss: 0.0401
Epoch [2/20] Batch [400] Loss: 0.0352
Epoch [2/20] Avg Loss: 0.0419
Epoch [3/20] Batch [0] Loss: 0.0437
Epoch [3/20] Batch [100] Loss: 0.0453
Epoch [3/20] Batch [200] Loss: 0.0350
Epoch [3/20] Batch [300] Loss: 0.0338
Epoch [3/20] Batch [400] Loss: 0.0354
Epoch [3/20] Avg Loss: 0.0365
Epoch [4/20] Batch [0] Loss: 0.0337
Epoch [4/20] Batch [100] Loss: 0.0350
Epoch [4/20] Batch [200] Loss: 0.0373
Epoch [4/20] Batch [300] Loss: 0.0345
Epoch [4/20] Batch [400] Loss: 0.0337
Epoch [4/20] Avg Loss: 0.0341
Epoch [5/20] Batch [0] Loss: 0.0347
Epoch [5/20] Batch [100] Loss: 0.0324
Epoch [5/20] Batch [200] Loss: 0.0337
Epoch [5/20] Batch [300] Loss: 0.0331
Epoch [5/20] Batch [400] Loss: 0.0317
Epoch [5/20] Avg Loss: 0.0323
Model saved at epoch 5
Epoch [6/20] Batch [0] Loss: 0.0283
Epoch [6/20] Batch [100] Loss: 0.0331
Epoch [6/20] Batch [200] Loss: 0.0282
Epoch [6/20] Batch [300] Loss: 0.0285
Epoch [6/20] Batch [400] Loss: 0.0303
Epoch [6/20] Avg Loss: 0.0311
Epoch [7/20] Batch [0] Loss: 0.0310
Epoch [7/20] Batch [100] Loss: 0.0353
Epoch [7/20] Batch [200] Loss: 0.0261
Epoch [7/20] Batch [300] Loss: 0.0282
Epoch [7/20] Batch [400] Loss: 0.0301
Epoch [7/20] Avg Loss: 0.0305
Epoch [8/20] Batch [0] Loss: 0.0346
Epoch [8/20] Batch [100] Loss: 0.0242
Epoch [8/20] Batch [200] Loss: 0.0279
Epoch [8/20] Batch [300] Loss: 0.0303
Epoch [8/20] Batch [400] Loss: 0.0294
Epoch [8/20] Avg Loss: 0.0297
Epoch [9/20] Batch [0] Loss: 0.0323
Epoch [9/20] Batch [100] Loss: 0.0324
```

```
Epoch [9/20] Batch [200] Loss: 0.0249
Epoch [9/20] Batch [300] Loss: 0.0320
Epoch [9/20] Batch [400] Loss: 0.0265
Epoch [9/20] Avg Loss: 0.0294
Epoch [10/20] Batch [0] Loss: 0.0296
Epoch [10/20] Batch [100] Loss: 0.0313
Epoch [10/20] Batch [200] Loss: 0.0312
Epoch [10/20] Batch [300] Loss: 0.0299
Epoch [10/20] Batch [400] Loss: 0.0261
Epoch [10/20] Avg Loss: 0.0289
Model saved at epoch 10
Epoch [11/20] Batch [0] Loss: 0.0292
Epoch [11/20] Batch [100] Loss: 0.0278
Epoch [11/20] Batch [200] Loss: 0.0272
Epoch [11/20] Batch [300] Loss: 0.0280
Epoch [11/20] Batch [400] Loss: 0.0281
Epoch [11/20] Avg Loss: 0.0283
Epoch [12/20] Batch [0] Loss: 0.0250
Epoch [12/20] Batch [100] Loss: 0.0268
Epoch [12/20] Batch [200] Loss: 0.0300
Epoch [12/20] Batch [300] Loss: 0.0246
Epoch [12/20] Batch [400] Loss: 0.0308
Epoch [12/20] Avg Loss: 0.0281
Epoch [13/20] Batch [0] Loss: 0.0224
Epoch [13/20] Batch [100] Loss: 0.0236
Epoch [13/20] Batch [200] Loss: 0.0284
Epoch [13/20] Batch [300] Loss: 0.0268
Epoch [13/20] Batch [400] Loss: 0.0277
Epoch [13/20] Avg Loss: 0.0276
Epoch [14/20] Batch [0] Loss: 0.0248
Epoch [14/20] Batch [100] Loss: 0.0274
Epoch [14/20] Batch [200] Loss: 0.0266
Epoch [14/20] Batch [300] Loss: 0.0299
Epoch [14/20] Batch [400] Loss: 0.0233
Epoch [14/20] Avg Loss: 0.0274
Epoch [15/20] Batch [0] Loss: 0.0267
Epoch [15/20] Batch [100] Loss: 0.0283
Epoch [15/20] Batch [200] Loss: 0.0291
Epoch [15/20] Batch [300] Loss: 0.0254
Epoch [15/20] Batch [400] Loss: 0.0287
Epoch [15/20] Avg Loss: 0.0275
Model saved at epoch 15
Epoch [16/20] Batch [0] Loss: 0.0265
Epoch [16/20] Batch [100] Loss: 0.0257
Epoch [16/20] Batch [200] Loss: 0.0332
Epoch [16/20] Batch [300] Loss: 0.0244
Epoch [16/20] Batch [400] Loss: 0.0266
Epoch [16/20] Avg Loss: 0.0274
Epoch [17/20] Batch [0] Loss: 0.0223
```

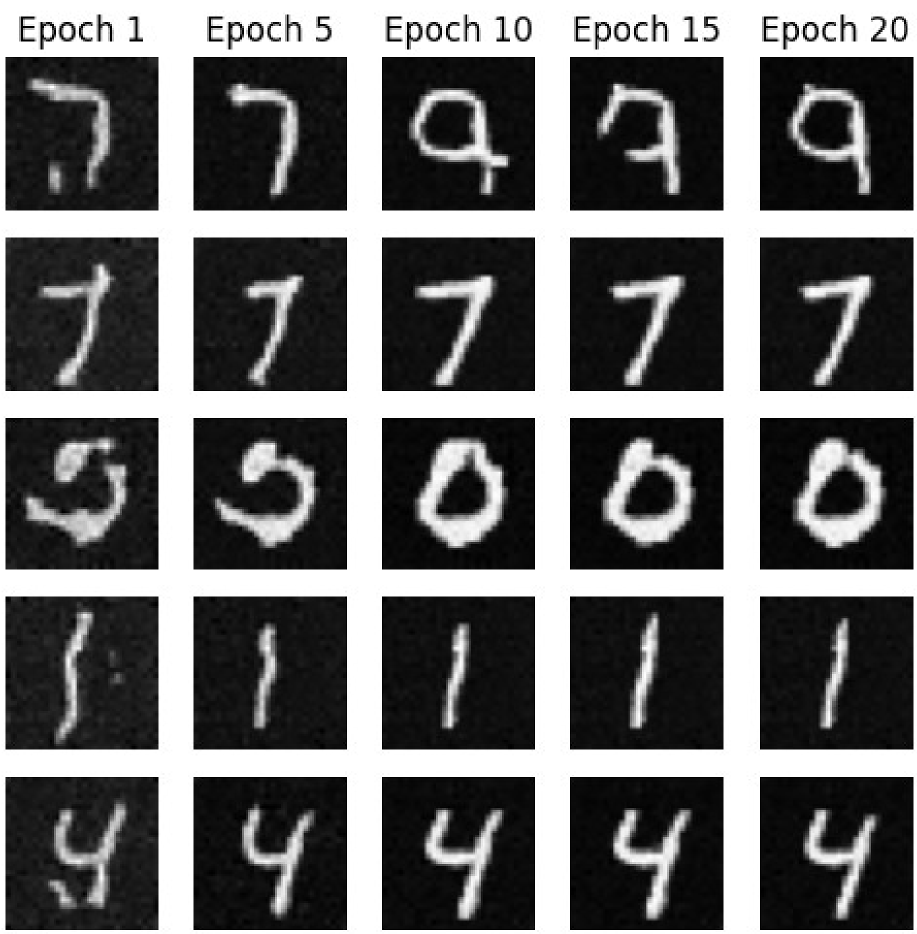
```
Epoch [17/20] Batch [100] Loss: 0.0240
Epoch [17/20] Batch [200] Loss: 0.0277
Epoch [17/20] Batch [300] Loss: 0.0245
Epoch [17/20] Batch [400] Loss: 0.0303
Epoch [17/20] Avg Loss: 0.0269
Epoch [18/20] Batch [0] Loss: 0.0262
Epoch [18/20] Batch [100] Loss: 0.0266
Epoch [18/20] Batch [200] Loss: 0.0260
Epoch [18/20] Batch [300] Loss: 0.0260
Epoch [18/20] Batch [400] Loss: 0.0248
Epoch [18/20] Avg Loss: 0.0270
Epoch [19/20] Batch [0] Loss: 0.0260
Epoch [19/20] Batch [100] Loss: 0.0246
Epoch [19/20] Batch [200] Loss: 0.0279
Epoch [19/20] Batch [300] Loss: 0.0250
Epoch [19/20] Batch [400] Loss: 0.0252
Epoch [19/20] Avg Loss: 0.0268
Epoch [20/20] Batch [0] Loss: 0.0279
Epoch [20/20] Batch [100] Loss: 0.0248
Epoch [20/20] Batch [200] Loss: 0.0318
Epoch [20/20] Batch [300] Loss: 0.0247
Epoch [20/20] Batch [400] Loss: 0.0264
Epoch [20/20] Avg Loss: 0.0267
Model saved at epoch 20
Training complete!
```



Training Loss over Epochs
Average Loss
Epoch
0.065
0.060
0.055
0.050
0.045
0.040
0.035
0.030
0.025
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


```
def visualize_images_with_titles(images: torch.Tensor, column_names:
list[str]):
    num_images, num_columns = images.shape[0], len(column_names)
    fig, axes = plt.subplots(num_images, num_columns,
figsize=(num_columns,num_images))

    for i, axr in enumerate(axes):
        for j, axc in enumerate(axr):
            img = images[i,j].cpu().numpy()

            axc.imshow(img, cmap='gray')
            axc.axis('off')

            if i == 0:
                axc.set_title(column_names[j])

    plt.tight_layout(pad=1)
    plt.show()

unet = TimeConditionalUNet(in_channels=1, num_classes=1,
num_hiddens=64).to(device)
```

```
model = DDPM(unet=unet, betas=(1e-4, 0.02), num_ts=300).to(device)

epoch_list = [1] + [i for i in range(5, epochs+1, 5)]
sampled_images_list = []
titles = []

for i in epoch_list:
    model.load_state_dict(torch.load(f"./models/model_epoch_{i}.pth"))
    sampled = ddpm_sample(unet=model.unet,
ddpm_schedule=model.ddpm_schedule, img_wh=(28, 28),
num_ts=model.num_ts, seed=2)
    sampled_images_list.append(sampled)
    titles.append(f'Epoch {i}')

all_samples = torch.hstack(sampled_images_list)

axes = visualize_images_with_titles(all_samples, titles)
```



Implementing class-conditioned UNet

```
device = "cuda" if torch.cuda.is_available() else "cpu"

class FCBlock(nn.Module):
    def __init__(self, in_channels: int, out_channels: int):
        super().__init__()
        self.fc1 = nn.Linear(in_channels, out_channels)
        self.gelu = nn.GELU()
        self.fc2 = nn.Linear(out_channels, out_channels)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.fc1(x)
        x = self.gelu(x)
        x = self.fc2(x)
        return x.unsqueeze(-1).unsqueeze(-1)

class ClassConditionalUNet(nn.Module):
    def __init__(
        self,
        in_channels: int,
        num_classes: int,
        num_hiddens: int,
    ):
        super().__init__()
        self.convblock1 = ConvBlock(in_channels, num_hiddens)
        self.convblock2 = ConvBlock(2*num_hiddens, num_hiddens)
        self.downblock1 = DownBlock(num_hiddens, num_hiddens)
        self.downblock2 = DownBlock(num_hiddens, 2*num_hiddens)
        self.flatten    = Flatten()
        self.unflatten  = Unflatten(2*num_hiddens)
        self.upblock1   = UpBlock(4*num_hiddens, num_hiddens)
        self.upblock2   = UpBlock(2*num_hiddens, num_hiddens)
        self.conv       = nn.Conv2d(num_hiddens, 1, kernel_size=3,
stride=1, padding=1)
        self.convblock1 = ConvBlock(in_channels, num_hiddens)
        self.convblock2 = ConvBlock(2*num_hiddens, num_hiddens)
        self.downblock1 = DownBlock(num_hiddens, num_hiddens)
        self.downblock2 = DownBlock(num_hiddens, 2*num_hiddens)
        self.flatten    = Flatten()
        self.unflatten  = Unflatten(2*num_hiddens)
        self.upblock1   = UpBlock(4*num_hiddens, num_hiddens)
        self.upblock2   = UpBlock(2*num_hiddens, num_hiddens)
        self.conv       = nn.Conv2d(num_hiddens, 1, kernel_size=3,
stride=1, padding=1)
        self.fcblock1   = FCBlock(1, 2*num_hiddens)
        self.fcblock2   = FCBlock(1, num_hiddens)
        self.fcblock3   = FCBlock(num_classes, 2*num_hiddens)
        self.fcblock4   = FCBlock(num_classes,   num_hiddens)
```

```
    def forward(
        self,
        x: torch.Tensor,
        c: torch.Tensor,
        t: torch.Tensor,
        mask: torch.Tensor | None = None,
    ) -> torch.Tensor:
        """
        Args:
            x: (N, C, H, W) input tensor.
            c: (N,) int64 condition tensor.
            t: (N,) normalized time tensor.
            mask: (N,) mask tensor. If not None, mask out condition
when mask == 0.

        Returns:
            (N, C, H, W) output tensor.
        """
        assert x.shape[-2:] == (28, 28), "Expect input shape to be
(28, 28)."
        c = F.one_hot(c, num_classes=10).float()
        if mask is not None :
            c = c*mask.unsqueeze(-1)
        x = self.convblock1(x)
        x1 = x
        x = self.downblock1(x)
        x2 = x
        x = self.downblock2(x)
        x3 = x
        x = self.flatten(x)
        x = self.fcblock3(c)*self.unflatten(x) + self.fcblock1(t)
        x = torch.cat((x,x3),dim=1)
        x = self.fcblock4(c)*self.upblock1(x) + self.fcblock2(t)
        x = torch.cat((x,x2),dim=1)
        x = self.upblock2(x)
        x = torch.cat((x,x1),dim=1)
        x = self.convblock2(x)
        x = self.conv(x)
        return x

def ddpm_forward(
    unet: ClassConditionalUNet,
    ddpm_schedule: dict,
    x_0: torch.Tensor,
    c: torch.Tensor,
    p_uncond: float,
    num_ts: int,
) -> torch.Tensor:
    """Algorithm 1 of the DDPM paper.
```

```
    Args:
        unet: ClassConditionalUNet
        ddpm_schedule: dict
        x_0: (N, C, H, W) input tensor.
        c: (N,) int64 condition tensor.
        p_uncond: float, probability of unconditioning the condition.
        num_ts: int, number of timesteps.

    Returns:
        (,) diffusion loss.
    """
    unet.train()
    # YOUR CODE HERE.
    t = torch.randint(1,num_ts+1,size=(x_0.size(0),1),device="cuda")
    alpha_bar_t = ddpm_schedule["alpha_bars"][t.squeeze(-1)]
    alpha_bar_t = alpha_bar_t.view(-1,1,1,1)
    mask = (torch.rand(x_0.size(0), device='cuda') > p_uncond).float()
    e = torch.randn_like(x_0)
    x_t = torch.sqrt(alpha_bar_t)*x_0 + torch.sqrt(1-alpha_bar_t)*e
    e_hat = unet(x=x_t, t=t.float()/num_ts, c=c, mask=mask)
    loss = nn.MSELoss()
    return loss(e,e_hat)
    raise NotImplementedError()

@torch.inference_mode()
def ddpm_sample(
    unet: ClassConditionalUNet,
    ddpm_schedule: dict,
    c: torch.Tensor,
    img_wh: tuple[int, int],
    num_ts: int,
    guidance_scale: float = 5.0,
    seed: int = 0,
) -> tuple[torch.Tensor, torch.Tensor]:
    """Algorithm 2 of the DDPM paper with classifier-free guidance +
animation.

    Args:
        unet: ClassConditionalUNet
        ddpm_schedule: dict
        c: (N,) int64 condition tensor. Only for class-conditional
        img_wh: (H, W) output image width and height.
        num_ts: int, number of timesteps.
        guidance_scale: float, CFG scale.
        seed: int, random seed.

    Returns:
        (N, C, H, W) final sample.
        (N, T_animation, C, H, W) all intermediate states for
animation.
```

```
    """
    unet.eval()
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)

    betas = ddpm_schedule["betas"]
    alphas = ddpm_schedule["alphas"]
    alpha_bars = ddpm_schedule["alpha_bars"]

    N = c.size(0)
    x_t = torch.randn(N, 1, img_wh[0], img_wh[1], device='cuda')

    frames = [x_t.clone()]

    for t in range(num_ts, 0, -1):
        alpha_t = alphas[t]
        beta_t = betas[t]
        alpha_bar_t = alpha_bars[t]
        alpha_bar_t_1 = alpha_bars[t-1]

        if t > 1:
            z = torch.randn_like(x_t, device='cuda')
        else:
            z = torch.zeros_like(x_t)

        s = t / num_ts * torch.ones((N, 1), device="cuda")

        eps_u = unet(x=x_t, t=s, c=c, mask=torch.zeros(N,
device='cuda'))
        eps_c = unet(x=x_t, t=s, c=c, mask=torch.ones(N,
device='cuda'))
        eps = eps_u + guidance_scale * (eps_c - eps_u)

        x_0_hat = (x_t - torch.sqrt(torch.abs(1 - alpha_bar_t)) * eps)
/ torch.sqrt(torch.abs(alpha_bar_t))
        x_t = (
            torch.sqrt(torch.abs(alpha_bar_t_1)) * beta_t * x_0_hat
            + torch.sqrt(torch.abs(alpha_t)) * (1 - alpha_bar_t_1) *
x_t
        ) / (1 - alpha_bar_t + 1e-5) + torch.sqrt(torch.abs(beta_t)) *
z

        frames.append(x_t.clone())

    animation = torch.stack(frames, dim=1)

    return x_t, animation

## GIF Generation

import torchvision.utils as vutils
```

```
import torchvision.transforms.functional as TF

def normalize_per_frame(frame: torch.Tensor) -> torch.Tensor:
    min_val = frame.amin(dim=(1, 2), keepdim=True)
    max_val = frame.amax(dim=(1, 2), keepdim=True)
    return (frame - min_val) / (max_val - min_val + 1e-8)

def save_ddpm_animation(
    animation: torch.Tensor,
    filename: str = "ddpm_animation.gif",
    total_duration_sec: int = 1,
    max_frames: int = 50,
    freeze_last_frame_sec: float = 1.0,
):

    animation = animation.detach().cpu()
    N, T, C, H, W = animation.shape

    step = max(1, T // max_frames)
    frames = []
    for t in range(0, T, step):
        frame_batch = animation[:, t]  # (N, C, H, W)
        normalized = torch.stack([normalize_per_frame(img) for img in 
frame_batch])
        grid = vutils.make_grid(normalized, nrow=N, padding=2)
        grid = grid[0, :, :].unsqueeze(0)
        img = TF.to_pil_image(grid.squeeze(0), mode='L')
        img = img.convert('P')

        frames.append(img)

    anim_duration_ms = int(1000 * total_duration_sec / len(frames))

    freeze_frame = frames[-1]
    freeze_repeats = int(1000 * freeze_last_frame_sec / 
anim_duration_ms)
    frames.extend([freeze_frame] * freeze_repeats)

    frames[0].save( filename, save_all=True, append_images=frames[1:], 
duration=anim_duration_ms, loop=0,  optimize=False)
    print(f"Saved GIF to {filename} with {len(frames)} frames 
({anim_duration_ms} ms/frame).")

class DDPM(nn.Module):
    def __init__(
        self,
        unet: ClassConditionalUNet,
        betas: tuple[float, float] = (1e-4, 0.02),
        num_ts: int = 300,
        p_uncond: float = 0.1,
```

```
    ):
        super().__init__()
        self.unet = unet
        self.betas = betas
        self.num_ts = num_ts
        self.p_uncond = p_uncond
        self.ddpm_schedule = ddpm_schedule(betas[0], betas[1], num_ts)

    def forward(self, x: torch.Tensor, c: torch.Tensor) ->
torch.Tensor:
        """
        Args:
            x: (N, C, H, W) input tensor.
            c: (N,) int64 condition tensor.

        Returns:
            (,) diffusion loss.
        """
        return ddpm_forward(
            self.unet, self.ddpm_schedule, x, c, self.p_uncond,
self.num_ts
        )

    @torch.inference_mode()
    def sample(
        self,
        c: torch.Tensor,
        img_wh: tuple[int, int],
        guidance_scale: float = 5.0,
        seed: int = 0,
    ):
        return ddpm_sample(
            self.unet, self.ddpm_schedule, c, img_wh, self.num_ts,
guidance_scale, seed
        )
```

Training the Class-conditioned UNet

- Plot the loss curve
- Sample results on the test set

```
dataset = MNIST(root='data', download=True, transform=ToTensor(),
train=True)
dataloader = DataLoader(dataset, batch_size=128, shuffle=True)

# TODO
device = 'cuda' if torch.cuda.is_available() else 'cpu'

unet = ClassConditionalUNet(in_channels=1, num_classes=10,
num_hiddens=64).to(device)
```

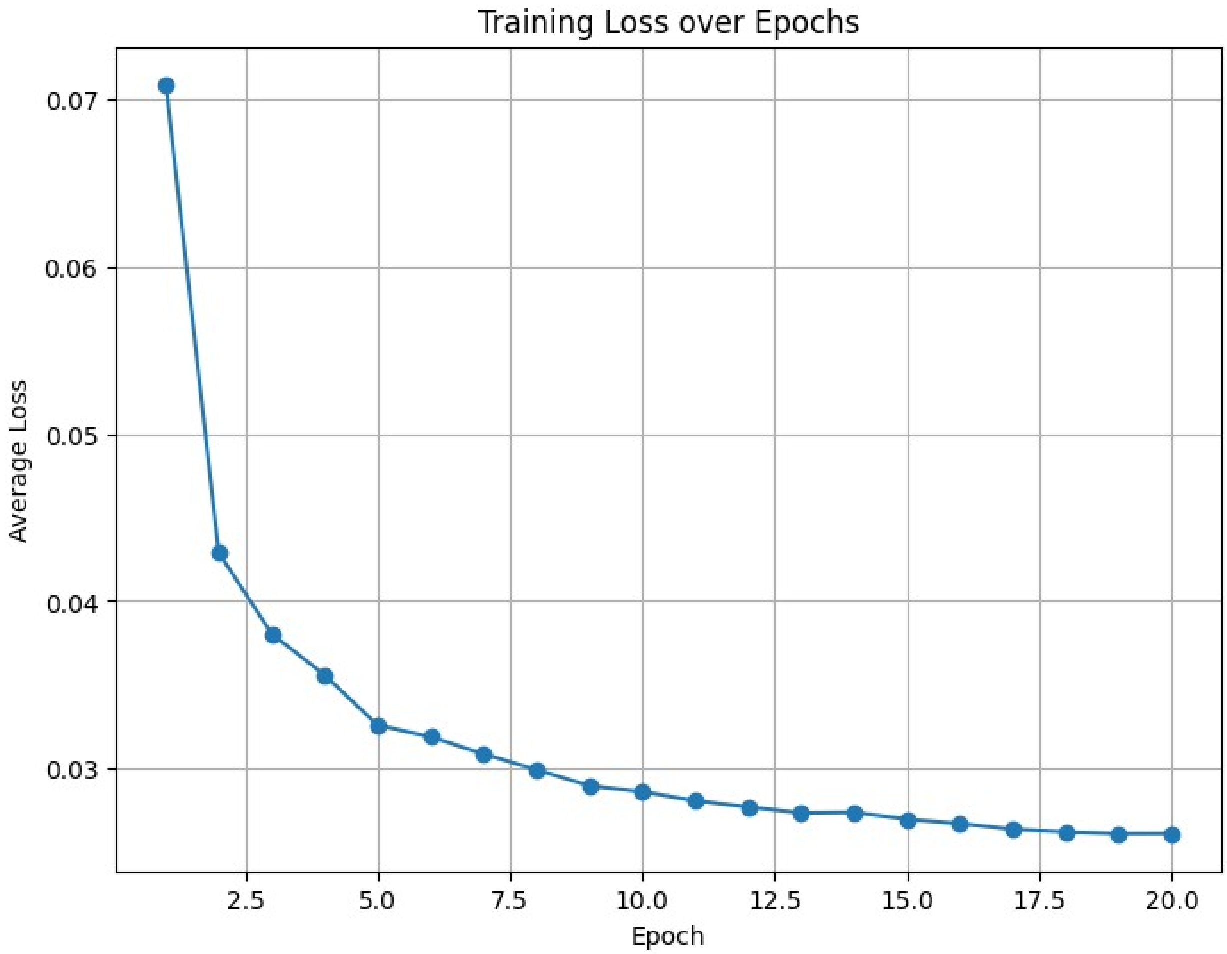
```
model = DDPM(unet=unet, betas=(1e-4, 0.02), num_ts=300,
p_uncond=0.1).to(device)

epochs = 20
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer,
gamma=0.1**(1/epochs))

epoch_losses = []
losses = []

for epoch in range(epochs):
    model.train()
    total_loss = 0

    for batch_idx, (x, labels) in enumerate(dataloader):
        x = x.to(device)
        labels = labels.to(device, dtype=torch.int64)

        optimizer.zero_grad()
        loss = model(x, labels)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

        if batch_idx % 2 == 0 :
            losses.append(loss.item())

        if batch_idx % 100 == 0:
            print(f"Epoch [{epoch+1}/{epochs}] Batch [{batch_idx}]
Loss: {loss.item():.4f}")

    scheduler.step()

    avg_loss = total_loss / len(dataloader)
    epoch_losses.append(avg_loss)

    print(f"Epoch [{epoch+1}/{epochs}] Avg Loss: {avg_loss:.4f}")

    if (epoch + 1) % 5 == 0 or epoch+1 == 1:
        torch.save(model.state_dict(),
f"./cls_cond_models/model_epoch_{epoch+1}.pth")
        print(f"Model saved at epoch {epoch+1}")

print("Training complete!")

plt.figure(figsize=(8,6))
plt.plot(range(1, epochs+1), epoch_losses, marker='o')
plt.xlabel('Epoch')
```

```python
plt.ylabel('Average Loss')
plt.title('Training Loss over Epochs')
plt.grid(True)
plt.show()

plt.figure(figsize=(8,6))
plt.plot(losses)
plt.xlabel('steps')
plt.yscale('log')
plt.ylabel('Loss')
plt.title('Training Loss over steps')
plt.grid(True)
plt.show()
```

```
Epoch [1/20] Batch [0] Loss: 1.2332
Epoch [1/20] Batch [100] Loss: 0.0624
Epoch [1/20] Batch [200] Loss: 0.0479
Epoch [1/20] Batch [300] Loss: 0.0478
Epoch [1/20] Batch [400] Loss: 0.0611
Epoch [1/20] Avg Loss: 0.0709
Model saved at epoch 1
Epoch [2/20] Batch [0] Loss: 0.0405
Epoch [2/20] Batch [100] Loss: 0.0489
Epoch [2/20] Batch [200] Loss: 0.0478
Epoch [2/20] Batch [300] Loss: 0.0416
Epoch [2/20] Batch [400] Loss: 0.0363
Epoch [2/20] Avg Loss: 0.0429
Epoch [3/20] Batch [0] Loss: 0.0482
Epoch [3/20] Batch [100] Loss: 0.0339
Epoch [3/20] Batch [200] Loss: 0.0382
Epoch [3/20] Batch [300] Loss: 0.0323
Epoch [3/20] Batch [400] Loss: 0.0369
Epoch [3/20] Avg Loss: 0.0381
Epoch [4/20] Batch [0] Loss: 0.0331
Epoch [4/20] Batch [100] Loss: 0.0434
Epoch [4/20] Batch [200] Loss: 0.0311
Epoch [4/20] Batch [300] Loss: 0.0336
Epoch [4/20] Batch [400] Loss: 0.0332
Epoch [4/20] Avg Loss: 0.0356
Epoch [5/20] Batch [0] Loss: 0.0363
Epoch [5/20] Batch [100] Loss: 0.0368
Epoch [5/20] Batch [200] Loss: 0.0300
Epoch [5/20] Batch [300] Loss: 0.0306
Epoch [5/20] Batch [400] Loss: 0.0305
Epoch [5/20] Avg Loss: 0.0326
Model saved at epoch 5
Epoch [6/20] Batch [0] Loss: 0.0309
Epoch [6/20] Batch [100] Loss: 0.0349
Epoch [6/20] Batch [200] Loss: 0.0293
Epoch [6/20] Batch [300] Loss: 0.0322
```

```
Epoch [6/20] Batch [400] Loss: 0.0275
Epoch [6/20] Avg Loss: 0.0319
Epoch [7/20] Batch [0] Loss: 0.0261
Epoch [7/20] Batch [100] Loss: 0.0315
Epoch [7/20] Batch [200] Loss: 0.0347
Epoch [7/20] Batch [300] Loss: 0.0356
Epoch [7/20] Batch [400] Loss: 0.0350
Epoch [7/20] Avg Loss: 0.0309
Epoch [8/20] Batch [0] Loss: 0.0268
Epoch [8/20] Batch [100] Loss: 0.0288
Epoch [8/20] Batch [200] Loss: 0.0263
Epoch [8/20] Batch [300] Loss: 0.0371
Epoch [8/20] Batch [400] Loss: 0.0309
Epoch [8/20] Avg Loss: 0.0299
Epoch [9/20] Batch [0] Loss: 0.0288
Epoch [9/20] Batch [100] Loss: 0.0273
Epoch [9/20] Batch [200] Loss: 0.0333
Epoch [9/20] Batch [300] Loss: 0.0241
Epoch [9/20] Batch [400] Loss: 0.0254
Epoch [9/20] Avg Loss: 0.0290
Epoch [10/20] Batch [0] Loss: 0.0276
Epoch [10/20] Batch [100] Loss: 0.0323
Epoch [10/20] Batch [200] Loss: 0.0251
Epoch [10/20] Batch [300] Loss: 0.0270
Epoch [10/20] Batch [400] Loss: 0.0301
Epoch [10/20] Avg Loss: 0.0286
Model saved at epoch 10
Epoch [11/20] Batch [0] Loss: 0.0252
Epoch [11/20] Batch [100] Loss: 0.0277
Epoch [11/20] Batch [200] Loss: 0.0297
Epoch [11/20] Batch [300] Loss: 0.0305
Epoch [11/20] Batch [400] Loss: 0.0303
Epoch [11/20] Avg Loss: 0.0281
Epoch [12/20] Batch [0] Loss: 0.0272
Epoch [12/20] Batch [100] Loss: 0.0241
Epoch [12/20] Batch [200] Loss: 0.0248
Epoch [12/20] Batch [300] Loss: 0.0274
Epoch [12/20] Batch [400] Loss: 0.0218
Epoch [12/20] Avg Loss: 0.0277
Epoch [13/20] Batch [0] Loss: 0.0262
Epoch [13/20] Batch [100] Loss: 0.0286
Epoch [13/20] Batch [200] Loss: 0.0265
Epoch [13/20] Batch [300] Loss: 0.0272
Epoch [13/20] Batch [400] Loss: 0.0289
Epoch [13/20] Avg Loss: 0.0274
Epoch [14/20] Batch [0] Loss: 0.0242
Epoch [14/20] Batch [100] Loss: 0.0251
Epoch [14/20] Batch [200] Loss: 0.0274
Epoch [14/20] Batch [300] Loss: 0.0252
```

```
Epoch [14/20] Batch [400] Loss: 0.0267
Epoch [14/20] Avg Loss: 0.0274
Epoch [15/20] Batch [0] Loss: 0.0233
Epoch [15/20] Batch [100] Loss: 0.0247
Epoch [15/20] Batch [200] Loss: 0.0266
Epoch [15/20] Batch [300] Loss: 0.0324
Epoch [15/20] Batch [400] Loss: 0.0263
Epoch [15/20] Avg Loss: 0.0270
Model saved at epoch 15
Epoch [16/20] Batch [0] Loss: 0.0296
Epoch [16/20] Batch [100] Loss: 0.0272
Epoch [16/20] Batch [200] Loss: 0.0261
Epoch [16/20] Batch [300] Loss: 0.0284
Epoch [16/20] Batch [400] Loss: 0.0247
Epoch [16/20] Avg Loss: 0.0267
Epoch [17/20] Batch [0] Loss: 0.0259
Epoch [17/20] Batch [100] Loss: 0.0243
Epoch [17/20] Batch [200] Loss: 0.0259
Epoch [17/20] Batch [300] Loss: 0.0299
Epoch [17/20] Batch [400] Loss: 0.0267
Epoch [17/20] Avg Loss: 0.0264
Epoch [18/20] Batch [0] Loss: 0.0270
Epoch [18/20] Batch [100] Loss: 0.0310
Epoch [18/20] Batch [200] Loss: 0.0228
Epoch [18/20] Batch [300] Loss: 0.0282
Epoch [18/20] Batch [400] Loss: 0.0261
Epoch [18/20] Avg Loss: 0.0262
Epoch [19/20] Batch [0] Loss: 0.0270
Epoch [19/20] Batch [100] Loss: 0.0246
Epoch [19/20] Batch [200] Loss: 0.0299
Epoch [19/20] Batch [300] Loss: 0.0238
Epoch [19/20] Batch [400] Loss: 0.0299
Epoch [19/20] Avg Loss: 0.0261
Epoch [20/20] Batch [0] Loss: 0.0269
Epoch [20/20] Batch [100] Loss: 0.0289
Epoch [20/20] Batch [200] Loss: 0.0252
Epoch [20/20] Batch [300] Loss: 0.0225
Epoch [20/20] Batch [400] Loss: 0.0297
Epoch [20/20] Avg Loss: 0.0261
Model saved at epoch 20
Training complete!
```



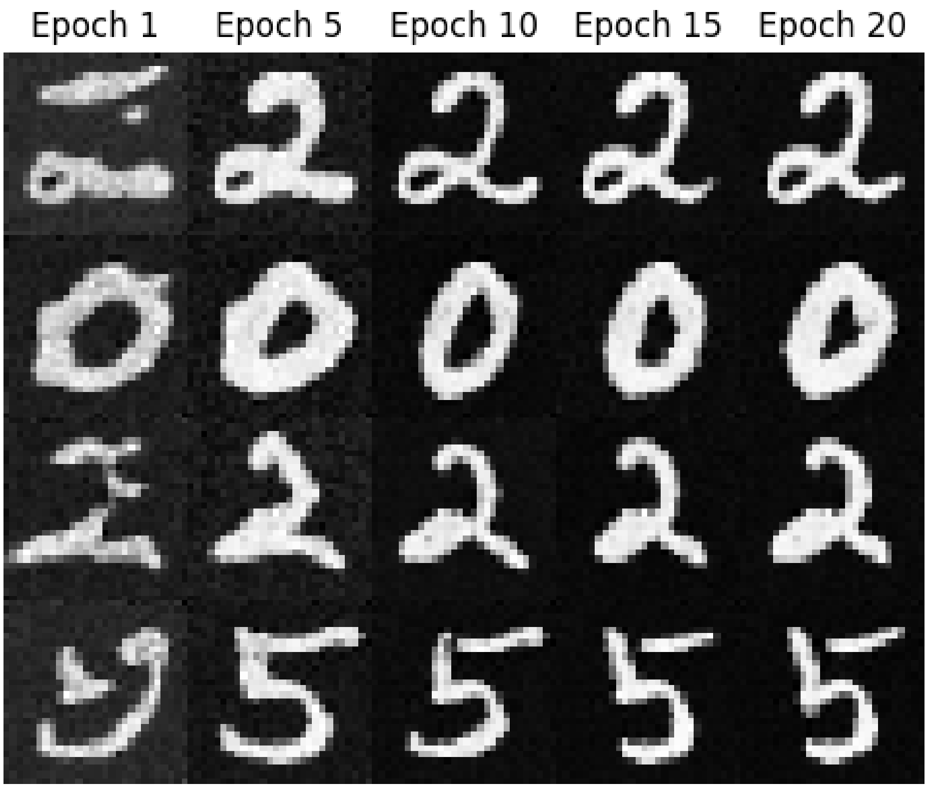
Training Loss over Epochs
0.07
0.06
0.05
0.04
0.03
Average Loss
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Epoch

Sampling + Extra Credits GIF Generation

```
def visualize_images_with_titles(images: torch.Tensor, column_names:
list[str]):
    num_images, num_columns = images.shape[0], len(column_names)
    fig, axes = plt.subplots(num_images, num_columns,
figsize=(0.95*num_columns,num_images))

    for i, axr in enumerate(axes):
        for j, axc in enumerate(axr):
            img = images[i,j].cpu().numpy()

            axc.imshow(img, cmap='gray')
            axc.axis('off')

            if i == 0:
                axc.set_title(column_names[j])

    plt.tight_layout(pad=0)
    plt.show()
```

```
unet = ClassConditionalUNet(in_channels=1, num_classes=10,
num_hiddens=64).to(device)
model = DDPM(unet=unet, betas=(1e-4, 0.02), num_ts=300,
p_uncond=0.1).to(device)

epoch_list = [1] + [i for i in range(5, epochs+1, 5)]
sampled_images_list = []
titles = []
c = torch.randint(0,10,size=(5,),device=device,dtype=torch.int64)
c = torch.tensor([2,0,2,5],dtype=torch.int64, device=device)

for i in epoch_list:

model.load_state_dict(torch.load(f"./cls_cond_models/model_epoch_{i}.p
th"))
    sampled, animation = model.sample(c=c,img_wh=(28, 28), seed=1)
    sampled_images_list.append(sampled)
    titles.append(f'Epoch {i}')
    # save_ddpm_animation(animation, f"ddpm_sampling_{i}.gif",
total_duration_sec=1)
    save_ddpm_animation(animation, f"ddpm_sampling_{i}.gif",
total_duration_sec=1, freeze_last_frame_sec=2,max_frames=50)

all_samples = torch.hstack(sampled_images_list)

axes = visualize_images_with_titles(all_samples, titles)
```

```
Saved GIF to ddpm_sampling_1.gif with 156 frames (19 ms/frame).
Saved GIF to ddpm_sampling_5.gif with 156 frames (19 ms/frame).
Saved GIF to ddpm_sampling_10.gif with 156 frames (19 ms/frame).
Saved GIF to ddpm_sampling_15.gif with 156 frames (19 ms/frame).
Saved GIF to ddpm_sampling_20.gif with 156 frames (19 ms/frame).
```



Extra Credits : Digit Classifier

```
from torchvision import transforms
import torch.optim as optim

class MNISTClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 64 * 7 * 7)
        x = F.relu(self.fc1(x))
        return self.fc2(x)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

transform = transforms.Compose([ transforms.ToTensor() ])
```

```
train_dataset = MNIST(root='./data', train=True, download=True,
transform=ToTensor())
test_dataset  = MNIST(root='./data', train=False, download=True,
transform=ToTensor())

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader  = DataLoader(test_dataset, batch_size=1000,
shuffle=False)

model = MNISTClassifier().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-3)

for epoch in range(20):
    model.train()
    running_loss = 0.0
    correct = 0

    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        _, preds = torch.max(outputs, 1)
        correct += (preds == labels).sum().item()

    acc = correct / len(train_loader.dataset)
    print(f"Epoch {epoch+1}, Loss: {running_loss:.4f}, Train Acc:
{acc:.4f}")
```

```
Epoch 1, Loss: 167.1055, Train Acc: 0.9465
Epoch 2, Loss: 46.5718, Train Acc: 0.9845
Epoch 3, Loss: 33.0304, Train Acc: 0.9892
Epoch 4, Loss: 23.6712, Train Acc: 0.9923
Epoch 5, Loss: 19.5774, Train Acc: 0.9932
Epoch 6, Loss: 14.6149, Train Acc: 0.9947
Epoch 7, Loss: 12.8404, Train Acc: 0.9955
Epoch 8, Loss: 9.5488, Train Acc: 0.9968
Epoch 9, Loss: 9.2326, Train Acc: 0.9968
Epoch 10, Loss: 7.7114, Train Acc: 0.9972
Epoch 11, Loss: 5.4378, Train Acc: 0.9980
Epoch 12, Loss: 5.3533, Train Acc: 0.9980
Epoch 13, Loss: 5.8896, Train Acc: 0.9978
Epoch 14, Loss: 3.4754, Train Acc: 0.9988
Epoch 15, Loss: 5.1009, Train Acc: 0.9980
```

```
Epoch 16, Loss: 3.5988, Train Acc: 0.9986
Epoch 17, Loss: 2.7178, Train Acc: 0.9988
Epoch 18, Loss: 2.0674, Train Acc: 0.9993
Epoch 19, Loss: 2.3118, Train Acc: 0.9991
Epoch 20, Loss: 3.8013, Train Acc: 0.9987
```

```
model.eval()
correct = 0

with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, preds = torch.max(outputs, 1)
        correct += (preds == labels).sum().item()

print(f"Test Accuracy: {correct / len(test_loader.dataset):.4f}")
```

```
Test Accuracy: 0.9907
```

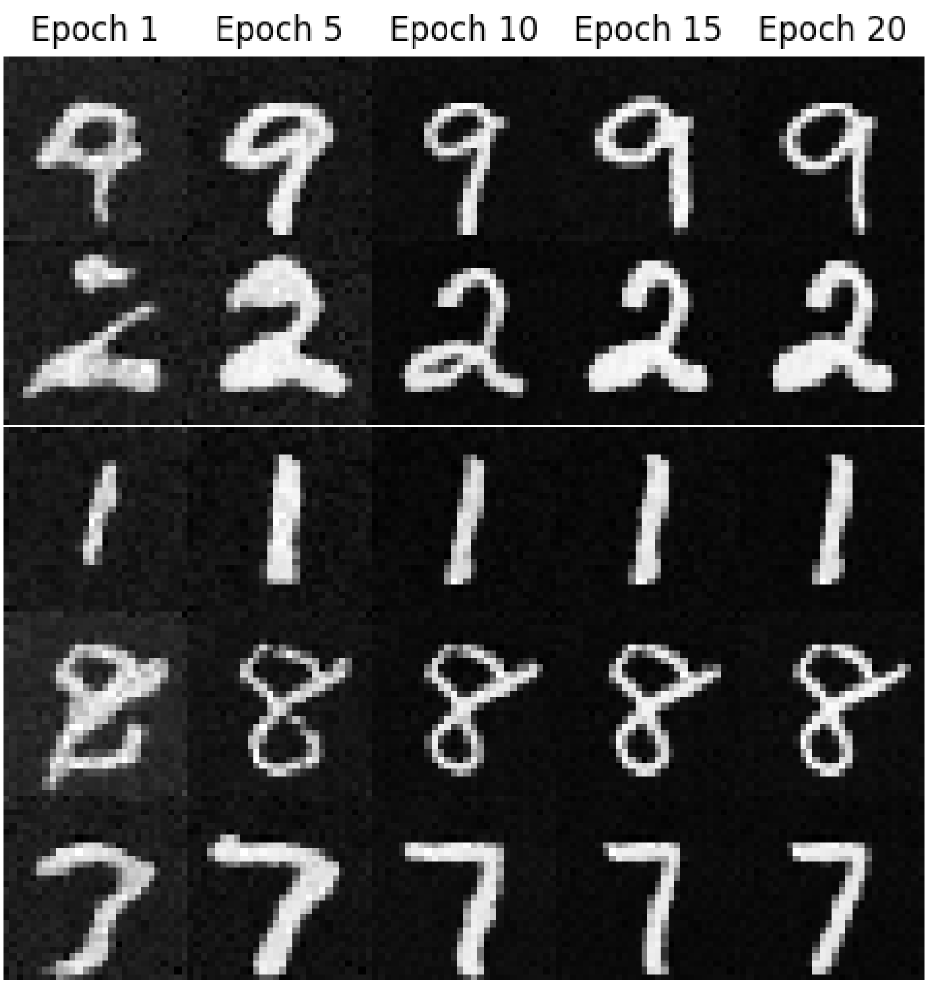
```
indices = torch.randint(0,1000,size=(5,))
c = preds[indices]
print("predicted : ",c)
print("correct : ",labels[indices])
```

```
predicted :  tensor([9, 2, 1, 8, 7], device='cuda:0')
correct :  tensor([9, 2, 1, 8, 7], device='cuda:0')
```

```
epochs = 20
unet = ClassConditionalUNet(in_channels=1, num_classes=10,
num_hiddens=64).to(device)
model = DDPM(unet=unet, betas=(1e-4, 0.02), num_ts=300,
p_uncond=0.1).to(device)

epoch_list = [1] + [i for i in range(5, epochs+1, 5)]
sampled_images_list = []
titles = []

for i in epoch_list:

model.load_state_dict(torch.load(f"./cls_cond_models/model_epoch_{i}.p
th"))
    sampled, animation = model.sample(c=c,img_wh=(28, 28), seed=1)
    sampled_images_list.append(sampled)
    titles.append(f'Epoch {i}')
    # save_ddpm_animation(animation, f"ddpm_sampling_{i}.gif",
total_duration_sec=1)
    # save_ddpm_animation(animation, f"extra_credits_{i}.gif",
total_duration_sec=1, freeze_last_frame_sec=2,max_frames=50)

all_samples = torch.hstack(sampled_images_list)
```

```
print(c)
axes = visualize_images_with_titles(all_samples, titles)
```

```
tensor([9, 2, 1, 8, 7], device='cuda:0')
```



Extra Credits : training on CIFAR 100

```
from torchvision.datasets import CIFAR100

dataset = CIFAR100(root="CIFAR100", download=True,
transform=ToTensor(), train=True)
dataloader = DataLoader(dataset, batch_size=128, shuffle=True)
device = 'cuda' if torch.cuda.is_available() else 'cpu'

class Unflatten(nn.Module):
    def __init__(self, in_channels: int):
        super().__init__()
        self.unflatten = nn.ConvTranspose2d(in_channels, in_channels,
kernel_size=8, stride=8, padding=0)
```

```
        self.bn      = nn.BatchNorm2d(in_channels)
        self.gelu    = nn.GELU()

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.unflatten(x)
        x = self.bn(x)
        x = self.gelu(x)
        return x

class TimeConditionalUNet(nn.Module):
    def __init__(
        self,
        in_channels: int,
        num_classes: int,
        num_hiddens: int,
    ):
        super().__init__()
        self.convblock1 = ConvBlock(in_channels, num_hiddens)
        self.convblock2 = ConvBlock(2*num_hiddens, num_hiddens)
        self.downblock1 = DownBlock(num_hiddens, num_hiddens)
        self.downblock2 = DownBlock(num_hiddens, 2*num_hiddens)
        self.flatten    = Flatten()
        self.unflatten  = Unflatten(2*num_hiddens)
        self.upblock1   = UpBlock(4*num_hiddens, num_hiddens)
        self.upblock2   = UpBlock(2*num_hiddens, num_hiddens)
        self.conv       = nn.Conv2d(num_hiddens, 1, kernel_size=3,
stride=1, padding=1)
        self.convblock1 = ConvBlock(in_channels, num_hiddens)
        self.convblock2 = ConvBlock(2*num_hiddens, num_hiddens)
        self.downblock1 = DownBlock(num_hiddens, num_hiddens)
        self.downblock2 = DownBlock(num_hiddens, 2*num_hiddens)
        self.flatten    = Flatten()
        self.unflatten  = Unflatten(2*num_hiddens)
        self.upblock1   = UpBlock(4*num_hiddens, num_hiddens)
        self.upblock2   = UpBlock(2*num_hiddens, num_hiddens)
        self.conv       = nn.Conv2d(num_hiddens, 3, kernel_size=3,
stride=1, padding=1)
        self.fcblock1   = FCBlock(1, 2*num_hiddens)
        self.fcblock2   = FCBlock(1, num_hiddens)

    def forward(
        self,
        x: torch.Tensor,
        t: torch.Tensor,
    ) -> torch.Tensor:
        """
        Args:
            x: (N, C, H, W) input tensor.
            t: (N,) normalized time tensor.
```

```
        Returns:
            (N, C, H, W) output tensor.
        """
        assert x.shape[-2:] == (32,32), "Expect input shape to be (28,
28)."
        x = self.convblock1(x)
        x1 = x
        x = self.downblock1(x)
        x2 = x
        x = self.downblock2(x)
        x3 = x
        x = self.flatten(x)
        x = self.unflatten(x) + self.fcblock1(t)
        x = torch.cat((x,x3),dim=1)
        x = self.upblock1(x) + self.fcblock2(t)
        x = torch.cat((x,x2),dim=1)
        x = self.upblock2(x)
        x = torch.cat((x,x1),dim=1)
        x = self.convblock2(x)
        x = self.conv(x)
        return x
        raise NotImplementedError()

@torch.inference_mode()
def ddpm_sample(
    unet: TimeConditionalUNet,
    ddpm_schedule: dict,
    img_wh: tuple[int, int],
    num_ts: int,
    seed: int = 0,
) -> torch.Tensor:
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    unet.eval()
    # YOUR CODE HERE.
    betas = ddpm_schedule["betas"]
    alphas = ddpm_schedule["alphas"]
    alpha_bars = ddpm_schedule["alpha_bars"]
    N   = 5
    x_t = torch.randn(N,3,img_wh[0],img_wh[1],device='cuda')
    for t in range(num_ts,0,-1):
        alpha_t = alphas[t]
        beta_t  = betas[t]
        alpha_bar_t = alpha_bars[t]
        alpha_bar_t_1 = alpha_bars[t-1]
        if t>1 :
            z = torch.randn_like(x_t,device='cuda')
        else :
            z = torch.zeros_like(x_t)
```

```
        x_0_hat = (x_t - torch.sqrt(torch.abs(1-
alpha_bar_t))*unet(x_t,
t/num_ts*torch.ones((N,1),device='cuda')))/torch.sqrt(torch.abs(alpha_
bar_t))
        x_t =   ( torch.sqrt(torch.abs(alpha_bar_t_1))*beta_t*x_0_hat
+ torch.sqrt(torch.abs(alpha_t))*(1-alpha_bar_t_1)*x_t )/((1-
alpha_bar_t)+1e-5) + torch.sqrt(torch.abs(beta_t))*z

    return x_t
    raise NotImplementedError()
```

Training

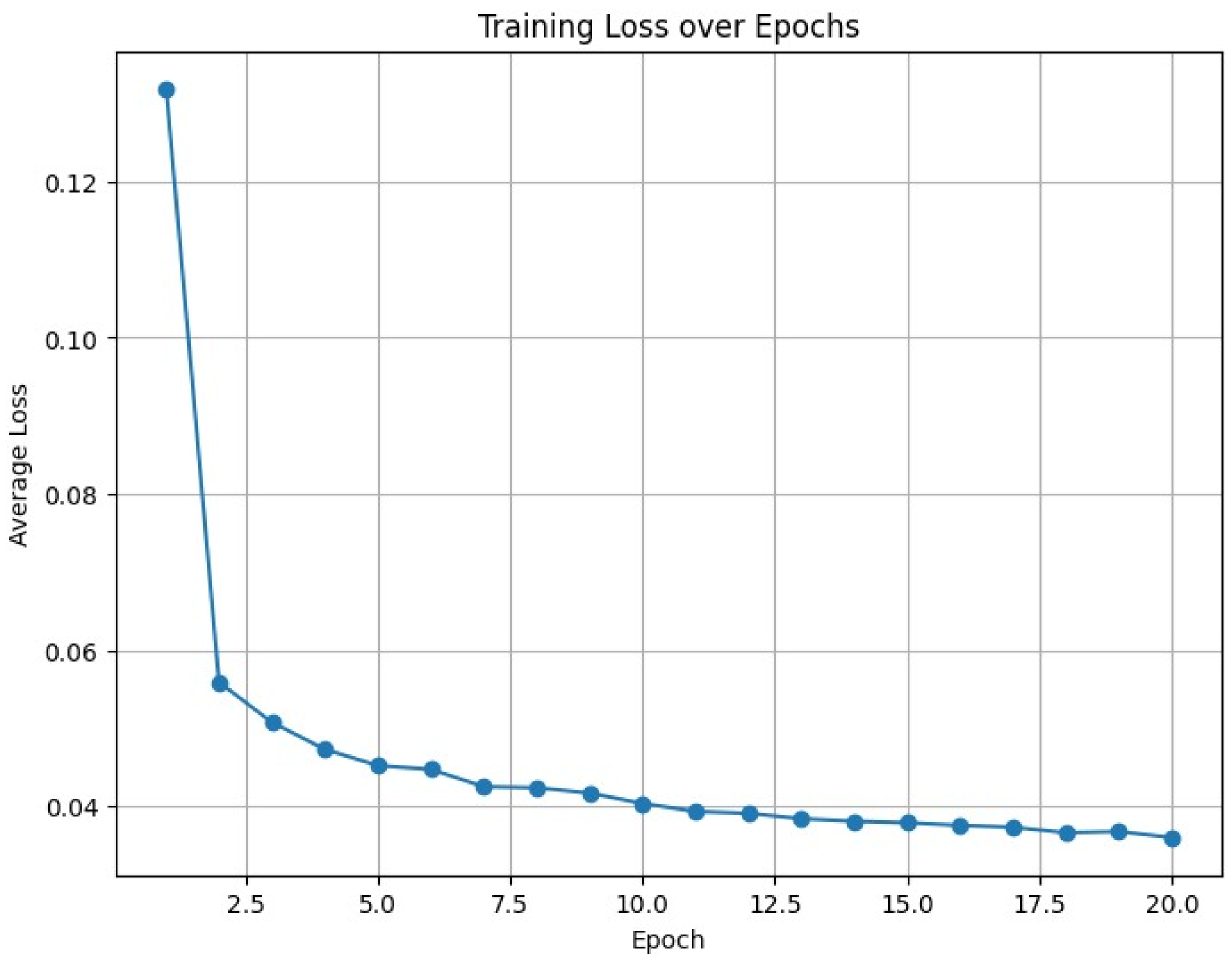
```
unet = TimeConditionalUNet(in_channels=3, num_classes=1,
num_hiddens=512).to(device)
model = DDPM(unet=unet, betas=(1e-4, 0.02), num_ts=300).to(device)

epochs = 20
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer,
gamma=0.1**(1/epochs))

epoch_losses = []

for epoch in range(epochs):
    model.train()
    total_loss = 0

    for batch_idx, (x, _) in enumerate(dataloader):
        x = x.to(device)

        optimizer.zero_grad()
        loss = model(x)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

        if batch_idx % 100 == 0:
            print(f"Epoch [{epoch+1}/{epochs}] Batch [{batch_idx}]
Loss: {loss.item():.4f}")

    scheduler.step()

    avg_loss = total_loss / len(dataloader)
    epoch_losses.append(avg_loss)

    print(f"Epoch [{epoch+1}/{epochs}] Avg Loss: {avg_loss:.4f}")

    if (epoch + 1) % 5 == 0 or epoch+1 == 1:
        torch.save(model.state_dict(),
```

```
f"./models/model_epoch_{epoch+1}.pth")
        print(f"Model saved at epoch {epoch+1}")

print("Training complete!")

plt.figure(figsize=(8,6))
plt.plot(range(1, epochs+1), epoch_losses, marker='o')
plt.xlabel('Epoch')
plt.ylabel('Average Loss')
plt.title('Training Loss over Epochs')
plt.grid(True)
plt.show()
```

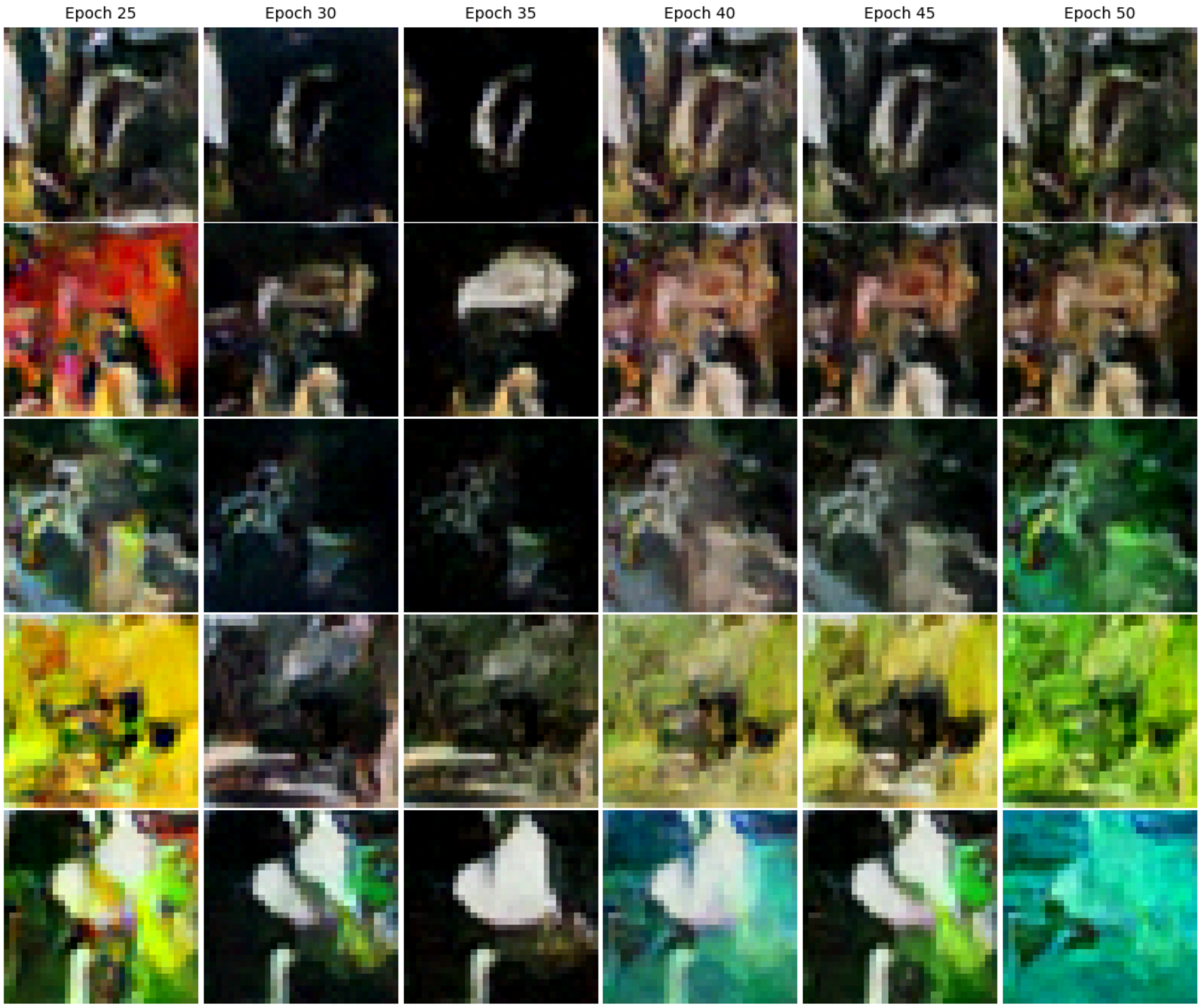
```
Epoch [1/20] Batch [0] Loss: 1.1461
Epoch [1/20] Batch [100] Loss: 0.0746
Epoch [1/20] Batch [200] Loss: 0.0737
Epoch [1/20] Batch [300] Loss: 0.0557
Epoch [1/20] Avg Loss: 0.1319
Model saved at epoch 1
Epoch [2/20] Batch [0] Loss: 0.0568
Epoch [2/20] Batch [100] Loss: 0.0509
Epoch [2/20] Batch [200] Loss: 0.0606
Epoch [2/20] Batch [300] Loss: 0.0488
Epoch [2/20] Avg Loss: 0.0559
Epoch [3/20] Batch [0] Loss: 0.0529
Epoch [3/20] Batch [100] Loss: 0.0382
Epoch [3/20] Batch [200] Loss: 0.0653
Epoch [3/20] Batch [300] Loss: 0.0632
Epoch [3/20] Avg Loss: 0.0507
Epoch [4/20] Batch [0] Loss: 0.0572
Epoch [4/20] Batch [100] Loss: 0.0473
Epoch [4/20] Batch [200] Loss: 0.0446
Epoch [4/20] Batch [300] Loss: 0.0543
Epoch [4/20] Avg Loss: 0.0473
Epoch [5/20] Batch [0] Loss: 0.0425
Epoch [5/20] Batch [100] Loss: 0.0369
Epoch [5/20] Batch [200] Loss: 0.0469
Epoch [5/20] Batch [300] Loss: 0.0611
Epoch [5/20] Avg Loss: 0.0452
Model saved at epoch 5
Epoch [6/20] Batch [0] Loss: 0.0455
Epoch [6/20] Batch [100] Loss: 0.0591
Epoch [6/20] Batch [200] Loss: 0.0432
Epoch [6/20] Batch [300] Loss: 0.0381
Epoch [6/20] Avg Loss: 0.0447
Epoch [7/20] Batch [0] Loss: 0.0381
Epoch [7/20] Batch [100] Loss: 0.0333
Epoch [7/20] Batch [200] Loss: 0.0442
Epoch [7/20] Batch [300] Loss: 0.0408
Epoch [7/20] Avg Loss: 0.0425
```

```
Epoch [8/20] Batch [0] Loss: 0.0346
Epoch [8/20] Batch [100] Loss: 0.0392
Epoch [8/20] Batch [200] Loss: 0.0354
Epoch [8/20] Batch [300] Loss: 0.0412
Epoch [8/20] Avg Loss: 0.0424
Epoch [9/20] Batch [0] Loss: 0.0358
Epoch [9/20] Batch [100] Loss: 0.0374
Epoch [9/20] Batch [200] Loss: 0.0469
Epoch [9/20] Batch [300] Loss: 0.0392
Epoch [9/20] Avg Loss: 0.0417
Epoch [10/20] Batch [0] Loss: 0.0424
Epoch [10/20] Batch [100] Loss: 0.0335
Epoch [10/20] Batch [200] Loss: 0.0422
Epoch [10/20] Batch [300] Loss: 0.0407
Epoch [10/20] Avg Loss: 0.0404
Model saved at epoch 10
Epoch [11/20] Batch [0] Loss: 0.0373
Epoch [11/20] Batch [100] Loss: 0.0391
Epoch [11/20] Batch [200] Loss: 0.0373
Epoch [11/20] Batch [300] Loss: 0.0387
Epoch [11/20] Avg Loss: 0.0394
Epoch [12/20] Batch [0] Loss: 0.0431
Epoch [12/20] Batch [100] Loss: 0.0343
Epoch [12/20] Batch [200] Loss: 0.0484
Epoch [12/20] Batch [300] Loss: 0.0418
Epoch [12/20] Avg Loss: 0.0391
Epoch [13/20] Batch [0] Loss: 0.0470
Epoch [13/20] Batch [100] Loss: 0.0390
Epoch [13/20] Batch [200] Loss: 0.0288
Epoch [13/20] Batch [300] Loss: 0.0421
Epoch [13/20] Avg Loss: 0.0384
Epoch [14/20] Batch [0] Loss: 0.0466
Epoch [14/20] Batch [100] Loss: 0.0351
Epoch [14/20] Batch [200] Loss: 0.0468
Epoch [14/20] Batch [300] Loss: 0.0285
Epoch [14/20] Avg Loss: 0.0381
Epoch [15/20] Batch [0] Loss: 0.0336
Epoch [15/20] Batch [100] Loss: 0.0439
Epoch [15/20] Batch [200] Loss: 0.0392
Epoch [15/20] Batch [300] Loss: 0.0357
Epoch [15/20] Avg Loss: 0.0379
Model saved at epoch 15
Epoch [16/20] Batch [0] Loss: 0.0472
Epoch [16/20] Batch [100] Loss: 0.0290
Epoch [16/20] Batch [200] Loss: 0.0353
Epoch [16/20] Batch [300] Loss: 0.0452
Epoch [16/20] Avg Loss: 0.0376
Epoch [17/20] Batch [0] Loss: 0.0419
Epoch [17/20] Batch [100] Loss: 0.0392
```

```
Epoch [17/20] Batch [200] Loss: 0.0346
Epoch [17/20] Batch [300] Loss: 0.0447
Epoch [17/20] Avg Loss: 0.0373
Epoch [18/20] Batch [0] Loss: 0.0305
Epoch [18/20] Batch [100] Loss: 0.0364
Epoch [18/20] Batch [200] Loss: 0.0396
Epoch [18/20] Batch [300] Loss: 0.0277
Epoch [18/20] Avg Loss: 0.0366
Epoch [19/20] Batch [0] Loss: 0.0322
Epoch [19/20] Batch [100] Loss: 0.0491
Epoch [19/20] Batch [200] Loss: 0.0275
Epoch [19/20] Batch [300] Loss: 0.0293
Epoch [19/20] Avg Loss: 0.0368
Epoch [20/20] Batch [0] Loss: 0.0437
Epoch [20/20] Batch [100] Loss: 0.0333
Epoch [20/20] Batch [200] Loss: 0.0298
Epoch [20/20] Batch [300] Loss: 0.0289
Epoch [20/20] Avg Loss: 0.0360
Model saved at epoch 20
Training complete!
```



```python
def visualize_images_with_titles(images: torch.Tensor, column_names: list[str]):
    assert images.dim() == 5, "Expected shape (rows, cols, 3, H, W)"
    rows, cols = images.shape[:2]
    fig, axes = plt.subplots(rows, cols, figsize=(1.8*cols, 1.8*rows))

    if rows == 1:
        axes = [axes]
    if cols == 1:
        axes = [[ax] for ax in axes]

    for i in range(rows):
        for j in range(cols):
            img = images[i, j].detach().cpu().permute(1, 2, 0)

            ax = axes[i][j]
            ax.imshow(img.clamp(0, 1))
            ax.axis('off')

            if i == 0:
                ax.set_title(column_names[j], fontsize=10)

    plt.tight_layout(pad=0.1)
    plt.show()

unet = TimeConditionalUNet(in_channels=3, num_classes=1, num_hiddens=256).to(device)
model = DDPM(unet=unet, betas=(1e-4, 0.02), num_ts=500).to(device)

epoch_list = [1] + [i for i in range(5, epochs+1, 5)]
epoch_list = [i for i in range(25, 50+1, 5)]
sampled_images_list = []
titles = []

for i in epoch_list:
    model.load_state_dict(torch.load(f"./models/model_epoch_{i}.pth"))
    sampled = ddpm_sample(unet=model.unet, ddpm_schedule=model.ddpm_schedule, img_wh=(32, 32), num_ts=model.num_ts, seed=5)
    sampled_images_list.append(sampled)
    titles.append(f'Epoch {i}')

all_samples = torch.stack(sampled_images_list,dim=1)

axes = visualize_images_with_titles(all_samples, titles)
```



Extra Credits : Modified Unet architecture

```
class Conv(nn.Module):
    def __init__(self, in_channels: int, out_channels: int):
        super().__init__()
        self.conv = nn.Conv2d(in_channels, out_channels,
kernel_size=3, stride=1, padding=1)
        self.bn   = nn.BatchNorm2d(out_channels)
        self.gelu = nn.GELU()

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.conv(x)
        x = self.bn(x)
        x = self.gelu(x)
        return x
        raise NotImplementedError()
```

```python
class DownConv(nn.Module):
    def __init__(self, in_channels: int, out_channels: int):
        super().__init__()
        self.down_conv = nn.Conv2d(in_channels, out_channels, 
kernel_size=3, stride=2, padding=1)
        self.bn   = nn.BatchNorm2d(out_channels)
        self.gelu = nn.GELU()

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.down_conv(x)
        x = self.bn(x)
        x = self.gelu(x)
        return x
        raise NotImplementedError()


class UpConv(nn.Module):
    def __init__(self, in_channels: int, out_channels: int):
        super().__init__()
        self.up_conv = nn.ConvTranspose2d(in_channels, out_channels, 
kernel_size=4, stride=2, padding=1)
        self.bn   = nn.BatchNorm2d(out_channels)
        self.gelu    = nn.GELU()

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.up_conv(x)
        x = self.bn(x)
        x = self.gelu(x)
        return x
        raise NotImplementedError()


class Flatten(nn.Module):
    def __init__(self):
        super().__init__()
        self.avg_pool = nn.AvgPool2d(kernel_size=7) # MODIFY according 
to image input size
        self.gelu     = nn.GELU()

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.avg_pool(x)
        x = self.gelu(x)
        return x
        raise NotImplementedError()


class Unflatten(nn.Module):
    def __init__(self, in_channels: int):
        super().__init__()
        self.unflatten = nn.ConvTranspose2d(in_channels, in_channels, 
```

```
kernel_size=7, stride=7, padding=0) # MODIFY according to image input
size
        self.bn   = nn.BatchNorm2d(in_channels)
        self.gelu    = nn.GELU()

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.unflatten(x)
        x = self.bn(x)
        x = self.gelu(x)
        return x
        raise NotImplementedError()


class ConvBlock(nn.Module):
    def __init__(self, in_channels: int, out_channels: int):
        super().__init__()
        self.conv1 = Conv(in_channels, out_channels)
        self.conv2 = Conv(out_channels, out_channels)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.conv1(x)
        res = x
        x = self.conv2(x)
        return x + res
        raise NotImplementedError()


class DownBlock(nn.Module):
    def __init__(self, in_channels: int, out_channels: int):
        super().__init__()
        self.down_conv = DownConv(in_channels, out_channels)
        self.conv      = ConvBlock(out_channels, out_channels)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.down_conv(x)
        x = self.conv(x)
        return x
        raise NotImplementedError()


class UpBlock(nn.Module):
    def __init__(self, in_channels: int, out_channels: int):
        super().__init__()
        self.up_conv = UpConv(in_channels, out_channels)
        self.conv    = ConvBlock(out_channels, out_channels)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.up_conv(x)
        x = self.conv(x)
        return x
```

```
        raise NotImplementedError()

class FCBlock(nn.Module):
    def __init__(self, in_channels: int, out_channels: int):
        super().__init__()
        self.fc1 = nn.Linear(in_channels, out_channels)
        self.gelu = nn.GELU()
        self.fc2 = nn.Linear(out_channels, out_channels)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.fc1(x)
        x = self.gelu(x)
        x = self.fc2(x)
        return x.unsqueeze(-1).unsqueeze(-1)

def sinusoidal_embedding(timesteps: torch.Tensor, dim: int):
    half_dim = dim // 2
    emb = torch.log(torch.tensor(10000, device='cuda')) / (half_dim -
1)
    emb = torch.exp(torch.arange(half_dim, device='cuda') * -emb)
    emb = timesteps * emb[None, :]
    emb = torch.cat((emb.sin(), emb.cos()), dim=1)
    return emb

class TimeEmbedding(nn.Module) :
    def __init__(self, out):
        super().__init__()
        self.fc1 = nn.Linear(out, out)
        self.gelu = nn.GELU()
        self.fc2 = nn.Linear(out, out)

    def forward(self, x) :
        x = sinusoidal_embedding(x,self.fc1.in_features)
        x = self.fc1(x)
        x = self.gelu(x)
        x = self.fc2(x)
        return x.unsqueeze(-1).unsqueeze(-1)

class TimeConditionalUNet(nn.Module):
    def __init__(
        self,
        in_channels: int,
        num_classes: int,
        num_hiddens: int,
    ):
        super().__init__()
        self.convblock1 = ConvBlock(in_channels, num_hiddens)
        self.convblock1a = ConvBlock(num_hiddens, num_hiddens) # Mod
        self.convblock1b = ConvBlock(num_hiddens, num_hiddens)
```

```
        self.convblock2 = ConvBlock(2*num_hiddens, num_hiddens)
        self.convblock2a = ConvBlock(num_hiddens, num_hiddens) # Mod
        self.convblock2b = ConvBlock(num_hiddens, num_hiddens)

        self.downblock1 = DownBlock(num_hiddens, num_hiddens)
        self.downblock2 = DownBlock(num_hiddens, 2*num_hiddens)
        self.flatten    = Flatten()
        self.unflatten  = Unflatten(2*num_hiddens)
        self.upblock1   = UpBlock(4*num_hiddens, num_hiddens)
        self.upblock2   = UpBlock(2*num_hiddens, num_hiddens)
        self.conv       = nn.Conv2d(num_hiddens, 1, kernel_size=3,
stride=1, padding=1)
        self.convblock1 = ConvBlock(in_channels, num_hiddens)
        self.convblock2 = ConvBlock(2*num_hiddens, num_hiddens)
        self.downblock1 = DownBlock(num_hiddens, num_hiddens)
        self.downblock2 = DownBlock(num_hiddens, 2*num_hiddens)
        self.flatten    = Flatten()
        self.unflatten  = Unflatten(2*num_hiddens)
        self.upblock1   = UpBlock(4*num_hiddens, num_hiddens)
        self.upblock2   = UpBlock(2*num_hiddens, num_hiddens)
        self.conv       = nn.Conv2d(num_hiddens, 1, kernel_size=3,
stride=1, padding=1) # MODIFY channels accordingly
        # self.fcblock1   = FCBlock(1, 2*num_hiddens)
        # self.fcblock2   = FCBlock(1, num_hiddens)
        # Sinusoidal Time embedding
        self.fcblock1   = TimeEmbedding(2*num_hiddens)
        self.fcblock2   = TimeEmbedding(num_hiddens)

    def forward(
        self,
        x: torch.Tensor,
        t: torch.Tensor,
    ) -> torch.Tensor:
        assert x.shape[-2:] == (28,28), "Expect input shape to be (28,
28)."
        x = self.convblock1(x)
        x = self.convblock1a(x)
        x = self.convblock1b(x)
        x1 = x

        x = self.downblock1(x)
        x2 = x
        x = self.downblock2(x)
        x3 = x
        # Neck
        x = self.flatten(x)
        x = self.unflatten(x) + self.fcblock1(t)
        # Upscaling
        x = torch.cat((x,x3),dim=1)
        x = self.upblock1(x) + self.fcblock2(t)
```

```
        x = torch.cat((x,x2),dim=1)
        x = self.upblock2(x)
        x = torch.cat((x,x1),dim=1)
        x = self.convblock2(x) + x1
        # Skip Connection :
        x = self.convblock2a(x)
        x = self.convblock2b(x)
        x = self.conv(x)
        return x
        raise NotImplementedError()
```

Training

```
unet = TimeConditionalUNet(in_channels=3, num_classes=1,
num_hiddens=128).to(device)
model = DDPM(unet=unet, betas=(1e-4, 0.02), num_ts=300).to(device)

epochs = 60
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer,
T_max=epochs, eta_min = 1e-8)

losses = []

for epoch in range(epochs):
    model.train()
    total_loss = 0

    for batch_idx, (x, _) in enumerate(dataloader):
        x = x.to(device)

        optimizer.zero_grad()
        loss = model(x)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

        if batch_idx % 100 == 0:
            print(f"Epoch [{epoch+1}/{epochs}] Batch [{batch_idx}]
Loss: {loss.item():.4f}")

    scheduler.step()

    avg_loss = total_loss / len(dataloader)
    losses.append(total_loss/x.size(0))

    print(f"Epoch [{epoch+1}/{epochs}] Avg Loss: {avg_loss:.4f}")

    if (epoch + 1) % 5 == 0 or epoch+1 == 1:
```

```python
        torch.save(model.state_dict(),
f"./models/model_epoch_{epoch+1}.pth")
        print(f"Model saved at epoch {epoch+1}")

print("Training complete!")

plt.figure(figsize=(8,6))
plt.plot(range(1, epochs+1), losses)
plt.xlabel('Epoch')
plt.ylabel('Average Loss')
plt.title('Training Loss over Epochs')
plt.grid(True)
plt.show()
```

```
Epoch [1/60] Batch [0] Loss: 2.2206
Epoch [1/60] Batch [100] Loss: 0.1032
Epoch [1/60] Batch [200] Loss: 0.1029
Epoch [1/60] Batch [300] Loss: 0.0690
Epoch [1/60] Avg Loss: 0.1248
Model saved at epoch 1
Epoch [2/60] Batch [0] Loss: 0.1096
Epoch [2/60] Batch [100] Loss: 0.0522
Epoch [2/60] Batch [200] Loss: 0.0604
Epoch [2/60] Batch [300] Loss: 0.0562
Epoch [2/60] Avg Loss: 0.0634
Epoch [3/60] Batch [0] Loss: 0.0653
Epoch [3/60] Batch [100] Loss: 0.0554
Epoch [3/60] Batch [200] Loss: 0.0393
Epoch [3/60] Batch [300] Loss: 0.0587
Epoch [3/60] Avg Loss: 0.0564
Epoch [4/60] Batch [0] Loss: 0.0518
Epoch [4/60] Batch [100] Loss: 0.0415
Epoch [4/60] Batch [200] Loss: 0.0470
Epoch [4/60] Batch [300] Loss: 0.0495
Epoch [4/60] Avg Loss: 0.0524
Epoch [5/60] Batch [0] Loss: 0.0419
Epoch [5/60] Batch [100] Loss: 0.0432
Epoch [5/60] Batch [200] Loss: 0.0415
Epoch [5/60] Batch [300] Loss: 0.0435
Epoch [5/60] Avg Loss: 0.0498
Model saved at epoch 5
Epoch [6/60] Batch [0] Loss: 0.0459
Epoch [6/60] Batch [100] Loss: 0.0435
Epoch [6/60] Batch [200] Loss: 0.0596
Epoch [6/60] Batch [300] Loss: 0.0513
Epoch [6/60] Avg Loss: 0.0475
Epoch [7/60] Batch [0] Loss: 0.0479
Epoch [7/60] Batch [100] Loss: 0.0476
Epoch [7/60] Batch [200] Loss: 0.0408
Epoch [7/60] Batch [300] Loss: 0.0440
```

```
Epoch [7/60] Avg Loss: 0.0458
Epoch [8/60] Batch [0] Loss: 0.0429
Epoch [8/60] Batch [100] Loss: 0.0394
Epoch [8/60] Batch [200] Loss: 0.0540
Epoch [8/60] Batch [300] Loss: 0.0491
Epoch [8/60] Avg Loss: 0.0466
Epoch [9/60] Batch [0] Loss: 0.0492
Epoch [9/60] Batch [100] Loss: 0.0476
Epoch [9/60] Batch [200] Loss: 0.0401
Epoch [9/60] Batch [300] Loss: 0.0467
Epoch [9/60] Avg Loss: 0.0442
Epoch [10/60] Batch [0] Loss: 0.0477
Epoch [10/60] Batch [100] Loss: 0.0456
Epoch [10/60] Batch [200] Loss: 0.0609
Epoch [10/60] Batch [300] Loss: 0.0451
Epoch [10/60] Avg Loss: 0.0433
Model saved at epoch 10
Epoch [11/60] Batch [0] Loss: 0.0458
Epoch [11/60] Batch [100] Loss: 0.0502
Epoch [11/60] Batch [200] Loss: 0.0540
Epoch [11/60] Batch [300] Loss: 0.0485
Epoch [11/60] Avg Loss: 0.0432
Epoch [12/60] Batch [0] Loss: 0.0334
Epoch [12/60] Batch [100] Loss: 0.0389
Epoch [12/60] Batch [200] Loss: 0.0503
Epoch [12/60] Batch [300] Loss: 0.0378
Epoch [12/60] Avg Loss: 0.0437
Epoch [13/60] Batch [0] Loss: 0.0324
Epoch [13/60] Batch [100] Loss: 0.0412
Epoch [13/60] Batch [200] Loss: 0.0490
Epoch [13/60] Batch [300] Loss: 0.0425
Epoch [13/60] Avg Loss: 0.0411
Epoch [14/60] Batch [0] Loss: 0.0419
Epoch [14/60] Batch [100] Loss: 0.0379
Epoch [14/60] Batch [200] Loss: 0.0456
Epoch [14/60] Batch [300] Loss: 0.0378
Epoch [14/60] Avg Loss: 0.0407
Epoch [15/60] Batch [0] Loss: 0.0418
Epoch [15/60] Batch [100] Loss: 0.0482
Epoch [15/60] Batch [200] Loss: 0.0359
Epoch [15/60] Batch [300] Loss: 0.0340
Epoch [15/60] Avg Loss: 0.0400
Model saved at epoch 15
Epoch [16/60] Batch [0] Loss: 0.0409
Epoch [16/60] Batch [100] Loss: 0.0471
Epoch [16/60] Batch [200] Loss: 0.0422
Epoch [16/60] Batch [300] Loss: 0.0512
Epoch [16/60] Avg Loss: 0.0394
Epoch [17/60] Batch [0] Loss: 0.0300
```

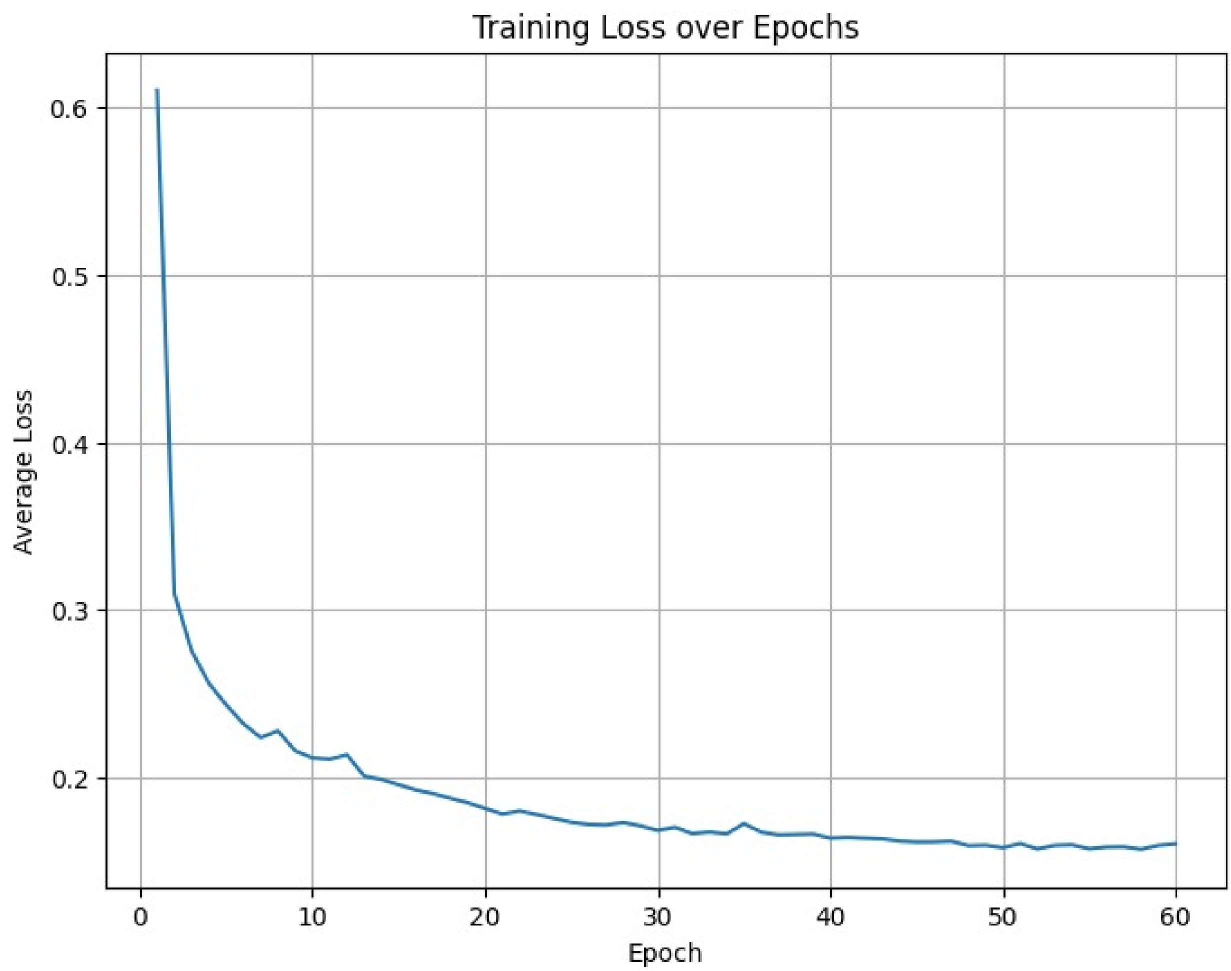
```
Epoch [17/60] Batch [100] Loss: 0.0413
Epoch [17/60] Batch [200] Loss: 0.0388
Epoch [17/60] Batch [300] Loss: 0.0273
Epoch [17/60] Avg Loss: 0.0390
Epoch [18/60] Batch [0] Loss: 0.0358
Epoch [18/60] Batch [100] Loss: 0.0315
Epoch [18/60] Batch [200] Loss: 0.0351
Epoch [18/60] Batch [300] Loss: 0.0355
Epoch [18/60] Avg Loss: 0.0384
Epoch [19/60] Batch [0] Loss: 0.0350
Epoch [19/60] Batch [100] Loss: 0.0410
Epoch [19/60] Batch [200] Loss: 0.0413
Epoch [19/60] Batch [300] Loss: 0.0333
Epoch [19/60] Avg Loss: 0.0379
Epoch [20/60] Batch [0] Loss: 0.0439
Epoch [20/60] Batch [100] Loss: 0.0353
Epoch [20/60] Batch [200] Loss: 0.0349
Epoch [20/60] Batch [300] Loss: 0.0395
Epoch [20/60] Avg Loss: 0.0372
Model saved at epoch 20
Epoch [21/60] Batch [0] Loss: 0.0436
Epoch [21/60] Batch [100] Loss: 0.0392
Epoch [21/60] Batch [200] Loss: 0.0451
Epoch [21/60] Batch [300] Loss: 0.0302
Epoch [21/60] Avg Loss: 0.0365
Epoch [22/60] Batch [0] Loss: 0.0290
Epoch [22/60] Batch [100] Loss: 0.0359
Epoch [22/60] Batch [200] Loss: 0.0350
Epoch [22/60] Batch [300] Loss: 0.0424
Epoch [22/60] Avg Loss: 0.0368
Epoch [23/60] Batch [0] Loss: 0.0452
Epoch [23/60] Batch [100] Loss: 0.0287
Epoch [23/60] Batch [200] Loss: 0.0373
Epoch [23/60] Batch [300] Loss: 0.0345
Epoch [23/60] Avg Loss: 0.0364
Epoch [24/60] Batch [0] Loss: 0.0262
Epoch [24/60] Batch [100] Loss: 0.0402
Epoch [24/60] Batch [200] Loss: 0.0328
Epoch [24/60] Batch [300] Loss: 0.0291
Epoch [24/60] Avg Loss: 0.0359
Epoch [25/60] Batch [0] Loss: 0.0345
Epoch [25/60] Batch [100] Loss: 0.0353
Epoch [25/60] Batch [200] Loss: 0.0301
Epoch [25/60] Batch [300] Loss: 0.0375
Epoch [25/60] Avg Loss: 0.0355
Model saved at epoch 25
Epoch [26/60] Batch [0] Loss: 0.0360
Epoch [26/60] Batch [100] Loss: 0.0352
Epoch [26/60] Batch [200] Loss: 0.0353
```

```
Epoch [26/60] Batch [300] Loss: 0.0337
Epoch [26/60] Avg Loss: 0.0352
Epoch [27/60] Batch [0] Loss: 0.0383
Epoch [27/60] Batch [100] Loss: 0.0408
Epoch [27/60] Batch [200] Loss: 0.0366
Epoch [27/60] Batch [300] Loss: 0.0461
Epoch [27/60] Avg Loss: 0.0351
Epoch [28/60] Batch [0] Loss: 0.0327
Epoch [28/60] Batch [100] Loss: 0.0383
Epoch [28/60] Batch [200] Loss: 0.0385
Epoch [28/60] Batch [300] Loss: 0.0337
Epoch [28/60] Avg Loss: 0.0354
Epoch [29/60] Batch [0] Loss: 0.0297
Epoch [29/60] Batch [100] Loss: 0.0356
Epoch [29/60] Batch [200] Loss: 0.0331
Epoch [29/60] Batch [300] Loss: 0.0401
Epoch [29/60] Avg Loss: 0.0350
Epoch [30/60] Batch [0] Loss: 0.0387
Epoch [30/60] Batch [100] Loss: 0.0330
Epoch [30/60] Batch [200] Loss: 0.0344
Epoch [30/60] Batch [300] Loss: 0.0355
Epoch [30/60] Avg Loss: 0.0345
Model saved at epoch 30
Epoch [31/60] Batch [0] Loss: 0.0331
Epoch [31/60] Batch [100] Loss: 0.0367
Epoch [31/60] Batch [200] Loss: 0.0388
Epoch [31/60] Batch [300] Loss: 0.0412
Epoch [31/60] Avg Loss: 0.0349
Epoch [32/60] Batch [0] Loss: 0.0303
Epoch [32/60] Batch [100] Loss: 0.0494
Epoch [32/60] Batch [200] Loss: 0.0304
Epoch [32/60] Batch [300] Loss: 0.0315
Epoch [32/60] Avg Loss: 0.0341
Epoch [33/60] Batch [0] Loss: 0.0342
Epoch [33/60] Batch [100] Loss: 0.0438
Epoch [33/60] Batch [200] Loss: 0.0307
Epoch [33/60] Batch [300] Loss: 0.0325
Epoch [33/60] Avg Loss: 0.0343
Epoch [34/60] Batch [0] Loss: 0.0297
Epoch [34/60] Batch [100] Loss: 0.0364
Epoch [34/60] Batch [200] Loss: 0.0340
Epoch [34/60] Batch [300] Loss: 0.0405
Epoch [34/60] Avg Loss: 0.0341
Epoch [35/60] Batch [0] Loss: 0.0258
Epoch [35/60] Batch [100] Loss: 0.0403
Epoch [35/60] Batch [200] Loss: 0.0344
Epoch [35/60] Batch [300] Loss: 0.0352
Epoch [35/60] Avg Loss: 0.0353
Model saved at epoch 35
```

```
Epoch [36/60] Batch [0] Loss: 0.0300
Epoch [36/60] Batch [100] Loss: 0.0310
Epoch [36/60] Batch [200] Loss: 0.0377
Epoch [36/60] Batch [300] Loss: 0.0375
Epoch [36/60] Avg Loss: 0.0343
Epoch [37/60] Batch [0] Loss: 0.0341
Epoch [37/60] Batch [100] Loss: 0.0419
Epoch [37/60] Batch [200] Loss: 0.0413
Epoch [37/60] Batch [300] Loss: 0.0332
Epoch [37/60] Avg Loss: 0.0339
Epoch [38/60] Batch [0] Loss: 0.0369
Epoch [38/60] Batch [100] Loss: 0.0250
Epoch [38/60] Batch [200] Loss: 0.0371
Epoch [38/60] Batch [300] Loss: 0.0260
Epoch [38/60] Avg Loss: 0.0340
Epoch [39/60] Batch [0] Loss: 0.0354
Epoch [39/60] Batch [100] Loss: 0.0346
Epoch [39/60] Batch [200] Loss: 0.0378
Epoch [39/60] Batch [300] Loss: 0.0307
Epoch [39/60] Avg Loss: 0.0340
Epoch [40/60] Batch [0] Loss: 0.0361
Epoch [40/60] Batch [100] Loss: 0.0322
Epoch [40/60] Batch [200] Loss: 0.0388
Epoch [40/60] Batch [300] Loss: 0.0339
Epoch [40/60] Avg Loss: 0.0335
Model saved at epoch 40
Epoch [41/60] Batch [0] Loss: 0.0342
Epoch [41/60] Batch [100] Loss: 0.0354
Epoch [41/60] Batch [200] Loss: 0.0377
Epoch [41/60] Batch [300] Loss: 0.0361
Epoch [41/60] Avg Loss: 0.0336
Epoch [42/60] Batch [0] Loss: 0.0340
Epoch [42/60] Batch [100] Loss: 0.0305
Epoch [42/60] Batch [200] Loss: 0.0323
Epoch [42/60] Batch [300] Loss: 0.0355
Epoch [42/60] Avg Loss: 0.0335
Epoch [43/60] Batch [0] Loss: 0.0377
Epoch [43/60] Batch [100] Loss: 0.0354
Epoch [43/60] Batch [200] Loss: 0.0414
Epoch [43/60] Batch [300] Loss: 0.0364
Epoch [43/60] Avg Loss: 0.0334
Epoch [44/60] Batch [0] Loss: 0.0381
Epoch [44/60] Batch [100] Loss: 0.0368
Epoch [44/60] Batch [200] Loss: 0.0347
Epoch [44/60] Batch [300] Loss: 0.0300
Epoch [44/60] Avg Loss: 0.0332
Epoch [45/60] Batch [0] Loss: 0.0261
Epoch [45/60] Batch [100] Loss: 0.0318
Epoch [45/60] Batch [200] Loss: 0.0353
```

```
Epoch [45/60] Batch [300] Loss: 0.0256
Epoch [45/60] Avg Loss: 0.0331
Model saved at epoch 45
Epoch [46/60] Batch [0] Loss: 0.0364
Epoch [46/60] Batch [100] Loss: 0.0353
Epoch [46/60] Batch [200] Loss: 0.0306
Epoch [46/60] Batch [300] Loss: 0.0308
Epoch [46/60] Avg Loss: 0.0331
Epoch [47/60] Batch [0] Loss: 0.0331
Epoch [47/60] Batch [100] Loss: 0.0385
Epoch [47/60] Batch [200] Loss: 0.0297
Epoch [47/60] Batch [300] Loss: 0.0369
Epoch [47/60] Avg Loss: 0.0332
Epoch [48/60] Batch [0] Loss: 0.0272
Epoch [48/60] Batch [100] Loss: 0.0277
Epoch [48/60] Batch [200] Loss: 0.0291
Epoch [48/60] Batch [300] Loss: 0.0269
Epoch [48/60] Avg Loss: 0.0326
Epoch [49/60] Batch [0] Loss: 0.0260
Epoch [49/60] Batch [100] Loss: 0.0385
Epoch [49/60] Batch [200] Loss: 0.0278
Epoch [49/60] Batch [300] Loss: 0.0282
Epoch [49/60] Avg Loss: 0.0327
Epoch [50/60] Batch [0] Loss: 0.0272
Epoch [50/60] Batch [100] Loss: 0.0326
Epoch [50/60] Batch [200] Loss: 0.0374
Epoch [50/60] Batch [300] Loss: 0.0292
Epoch [50/60] Avg Loss: 0.0324
Model saved at epoch 50
Epoch [51/60] Batch [0] Loss: 0.0348
Epoch [51/60] Batch [100] Loss: 0.0400
Epoch [51/60] Batch [200] Loss: 0.0269
Epoch [51/60] Batch [300] Loss: 0.0264
Epoch [51/60] Avg Loss: 0.0329
Epoch [52/60] Batch [0] Loss: 0.0293
Epoch [52/60] Batch [100] Loss: 0.0327
Epoch [52/60] Batch [200] Loss: 0.0303
Epoch [52/60] Batch [300] Loss: 0.0355
Epoch [52/60] Avg Loss: 0.0322
Epoch [53/60] Batch [0] Loss: 0.0297
Epoch [53/60] Batch [100] Loss: 0.0432
Epoch [53/60] Batch [200] Loss: 0.0335
Epoch [53/60] Batch [300] Loss: 0.0258
Epoch [53/60] Avg Loss: 0.0327
Epoch [54/60] Batch [0] Loss: 0.0275
Epoch [54/60] Batch [100] Loss: 0.0242
Epoch [54/60] Batch [200] Loss: 0.0316
Epoch [54/60] Batch [300] Loss: 0.0238
Epoch [54/60] Avg Loss: 0.0327
```

```
Epoch [55/60] Batch [0] Loss: 0.0309
Epoch [55/60] Batch [100] Loss: 0.0287
Epoch [55/60] Batch [200] Loss: 0.0315
Epoch [55/60] Batch [300] Loss: 0.0339
Epoch [55/60] Avg Loss: 0.0322
Model saved at epoch 55
Epoch [56/60] Batch [0] Loss: 0.0290
Epoch [56/60] Batch [100] Loss: 0.0307
Epoch [56/60] Batch [200] Loss: 0.0352
Epoch [56/60] Batch [300] Loss: 0.0273
Epoch [56/60] Avg Loss: 0.0325
Epoch [57/60] Batch [0] Loss: 0.0329
Epoch [57/60] Batch [100] Loss: 0.0309
Epoch [57/60] Batch [200] Loss: 0.0287
Epoch [57/60] Batch [300] Loss: 0.0253
Epoch [57/60] Avg Loss: 0.0325
Epoch [58/60] Batch [0] Loss: 0.0308
Epoch [58/60] Batch [100] Loss: 0.0274
Epoch [58/60] Batch [200] Loss: 0.0333
Epoch [58/60] Batch [300] Loss: 0.0334
Epoch [58/60] Avg Loss: 0.0322
Epoch [59/60] Batch [0] Loss: 0.0359
Epoch [59/60] Batch [100] Loss: 0.0411
Epoch [59/60] Batch [200] Loss: 0.0280
Epoch [59/60] Batch [300] Loss: 0.0240
Epoch [59/60] Avg Loss: 0.0327
Epoch [60/60] Batch [0] Loss: 0.0329
Epoch [60/60] Batch [100] Loss: 0.0276
Epoch [60/60] Batch [200] Loss: 0.0370
Epoch [60/60] Batch [300] Loss: 0.0370
Epoch [60/60] Avg Loss: 0.0328
Model saved at epoch 60
Training complete!
```

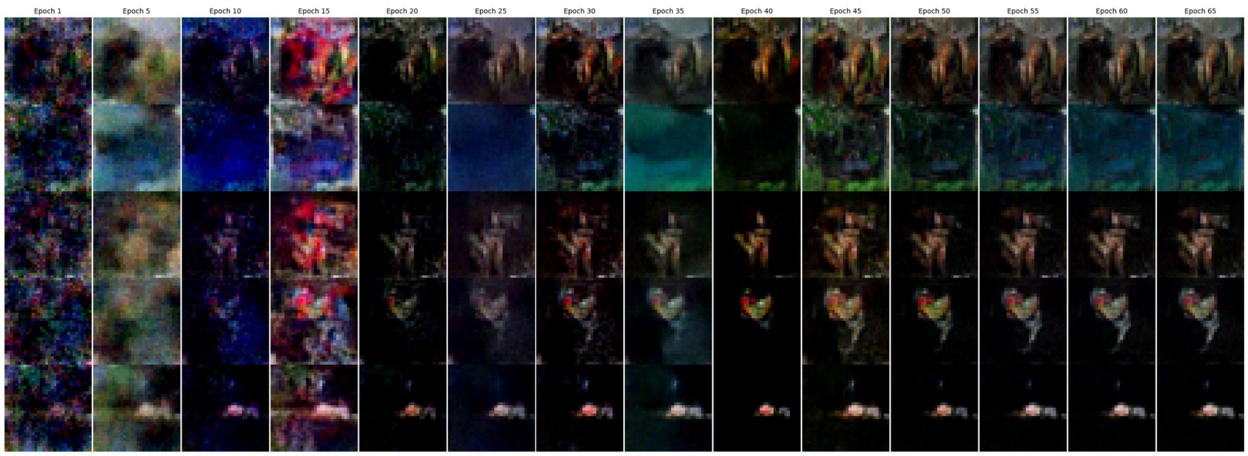



```
model.load_state_dict(torch.load(f"./models/model_epoch_{60}.pth"))
```

```
<All keys matched successfully>
```

```
unet = TimeConditionalUNet(in_channels=3, num_classes=1,
num_hiddens=128).to(device)
model = DDPM(unet=unet, betas=(1e-4, 0.02), num_ts=300).to(device)

epoch_list = [1] + [i for i in range(5, 70, 5)]
# epoch_list = [i for i in range(0, 50+1, 5)]
sampled_images_list = []
titles = []

for i in epoch_list:
    model.load_state_dict(torch.load(f"./models/model_epoch_{i}.pth"))
    sampled = ddpm_sample(unet=model.unet,
ddpm_schedule=model.ddpm_schedule, img_wh=(32, 32),
num_ts=model.num_ts, seed=0)
    sampled_images_list.append(sampled)
    titles.append(f'Epoch {i}')
```

```python
all_samples = torch.stack(sampled_images_list,dim=1)

axes = visualize_images_with_titles(all_samples, titles)
```



Rectified Flow :

```python
def rectified_flow_forward(
    unet: TimeConditionalUNet,
    x_0: torch.Tensor,
) -> torch.Tensor:
    unet.train()
    batch_size = x_0.size(0)

    x_1 = torch.randn_like(x_0, device=x_0.device)
    t = torch.rand(batch_size, 1, 1, 1, device=x_0.device)
    x_t = (1 - t) * x_0 + t * x_1
    true_flow = x_1 - x_0
    pred_flow = unet(x_t, t.squeeze(-1).squeeze(-1))
    loss = nn.MSELoss()
    return loss(pred_flow, true_flow)

@torch.inference_mode()
def rectified_flow_sample(
    unet: TimeConditionalUNet,
    img_wh: tuple[int, int],
    num_ts: int,
    seed: int = 0,
) -> torch.Tensor:
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    unet.eval()

    N = 5
    H, W = img_wh
    x_t = torch.randn(N, 1, H, W, device='cuda')
```

```
    dt = 1.0 / num_ts
    for i in range(num_ts):
        t = 1.0 - i * dt
        t_tensor = t * torch.ones((N, 1), device='cuda')
        f_theta = unet(x_t, t_tensor)
        x_t = x_t - dt * f_theta
    return x_t

class RF(nn.Module):
    def __init__(
        self,
        unet: TimeConditionalUNet,
        num_ts: int = 300,
        p_uncond: float = 0.1,
    ):
        super().__init__()
        self.unet = unet
        self.num_ts = num_ts
        self.p_uncond = p_uncond


    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return rectified_flow_forward( self.unet, x)

    @torch.inference_mode()
    def sample( self, img_wh: tuple[int, int], seed: int = 0,):
        return rectified_flow_sample( self.unet, img_wh, self.num_ts,
seed)

dataset = MNIST(root='data', download=True, transform=ToTensor(),
train=True)
dataloader = DataLoader(dataset, batch_size=150, shuffle=True)

device = 'cuda' if torch.cuda.is_available() else 'cpu'

unet = TimeConditionalUNet(in_channels=1, num_classes=1,
num_hiddens=64).to(device)
model = RF(unet=unet, num_ts=500).to(device)

epochs = 50
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer,
T_max=epochs, eta_min = 1e-8)

epoch_losses = []

for epoch in range(epochs):
    model.train()
    total_loss = 0

    for batch_idx, (x, _) in enumerate(dataloader):
```

```
        x = x.to(device)

        optimizer.zero_grad()
        loss = model(x)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

        if batch_idx % 100 == 0:
            print(f"Epoch [{epoch+1}/{epochs}] Batch [{batch_idx}]
Loss: {loss.item():.4f}")

    scheduler.step()

    avg_loss = total_loss / len(dataloader)
    epoch_losses.append(avg_loss)

    print(f"Epoch [{epoch+1}/{epochs}] Avg Loss: {avg_loss:.4f}")

    # Save model every 5 epochs
    if (epoch + 1) % 5 == 0 or epoch+1 == 1:
        torch.save(model.state_dict(),
f"./rf_models/model_epoch_{epoch+1}.pth")
        print(f"Model saved at epoch {epoch+1}")

print("Training complete!")

# Plotting loss curve
plt.figure(figsize=(8,6))
plt.plot(range(1, epochs+1), epoch_losses, marker='o')
plt.xlabel('Epoch')
plt.ylabel('Average Loss')
plt.title('Training Loss over Epochs')
plt.grid(True)
plt.show()
```

```
Epoch [1/50] Batch [0] Loss: 1.5580
Epoch [1/50] Batch [100] Loss: 0.1691
Epoch [1/50] Batch [200] Loss: 0.1328
Epoch [1/50] Batch [300] Loss: 0.1362
Epoch [1/50] Avg Loss: 0.1678
Model saved at epoch 1
Epoch [2/50] Batch [0] Loss: 0.1287
Epoch [2/50] Batch [100] Loss: 0.1258
Epoch [2/50] Batch [200] Loss: 0.1179
Epoch [2/50] Batch [300] Loss: 0.1114
Epoch [2/50] Avg Loss: 0.1133
Epoch [3/50] Batch [0] Loss: 0.1111
Epoch [3/50] Batch [100] Loss: 0.1062
```

```
Epoch [3/50] Batch [200] Loss: 0.1050
Epoch [3/50] Batch [300] Loss: 0.1131
Epoch [3/50] Avg Loss: 0.1051
Epoch [4/50] Batch [0] Loss: 0.1096
Epoch [4/50] Batch [100] Loss: 0.0995
Epoch [4/50] Batch [200] Loss: 0.0951
Epoch [4/50] Batch [300] Loss: 0.0994
Epoch [4/50] Avg Loss: 0.1017
Epoch [5/50] Batch [0] Loss: 0.1001
Epoch [5/50] Batch [100] Loss: 0.0984
Epoch [5/50] Batch [200] Loss: 0.0969
Epoch [5/50] Batch [300] Loss: 0.1056
Epoch [5/50] Avg Loss: 0.0988
Model saved at epoch 5
Epoch [6/50] Batch [0] Loss: 0.0980
Epoch [6/50] Batch [100] Loss: 0.0919
Epoch [6/50] Batch [200] Loss: 0.0949
Epoch [6/50] Batch [300] Loss: 0.1112
Epoch [6/50] Avg Loss: 0.0970
Epoch [7/50] Batch [0] Loss: 0.0961
Epoch [7/50] Batch [100] Loss: 0.1054
Epoch [7/50] Batch [200] Loss: 0.1019
Epoch [7/50] Batch [300] Loss: 0.1021
Epoch [7/50] Avg Loss: 0.0955
Epoch [8/50] Batch [0] Loss: 0.0907
Epoch [8/50] Batch [100] Loss: 0.0953
Epoch [8/50] Batch [200] Loss: 0.0985
Epoch [8/50] Batch [300] Loss: 0.0880
Epoch [8/50] Avg Loss: 0.0941
Epoch [9/50] Batch [0] Loss: 0.0970
Epoch [9/50] Batch [100] Loss: 0.0927
Epoch [9/50] Batch [200] Loss: 0.0994
Epoch [9/50] Batch [300] Loss: 0.0896
Epoch [9/50] Avg Loss: 0.0932
Epoch [10/50] Batch [0] Loss: 0.0919
Epoch [10/50] Batch [100] Loss: 0.0876
Epoch [10/50] Batch [200] Loss: 0.0915
Epoch [10/50] Batch [300] Loss: 0.0942
Epoch [10/50] Avg Loss: 0.0923
Model saved at epoch 10
Epoch [11/50] Batch [0] Loss: 0.0990
Epoch [11/50] Batch [100] Loss: 0.0907
Epoch [11/50] Batch [200] Loss: 0.0907
Epoch [11/50] Batch [300] Loss: 0.0957
Epoch [11/50] Avg Loss: 0.0916
Epoch [12/50] Batch [0] Loss: 0.0903
Epoch [12/50] Batch [100] Loss: 0.0942
Epoch [12/50] Batch [200] Loss: 0.0897
Epoch [12/50] Batch [300] Loss: 0.0869
```

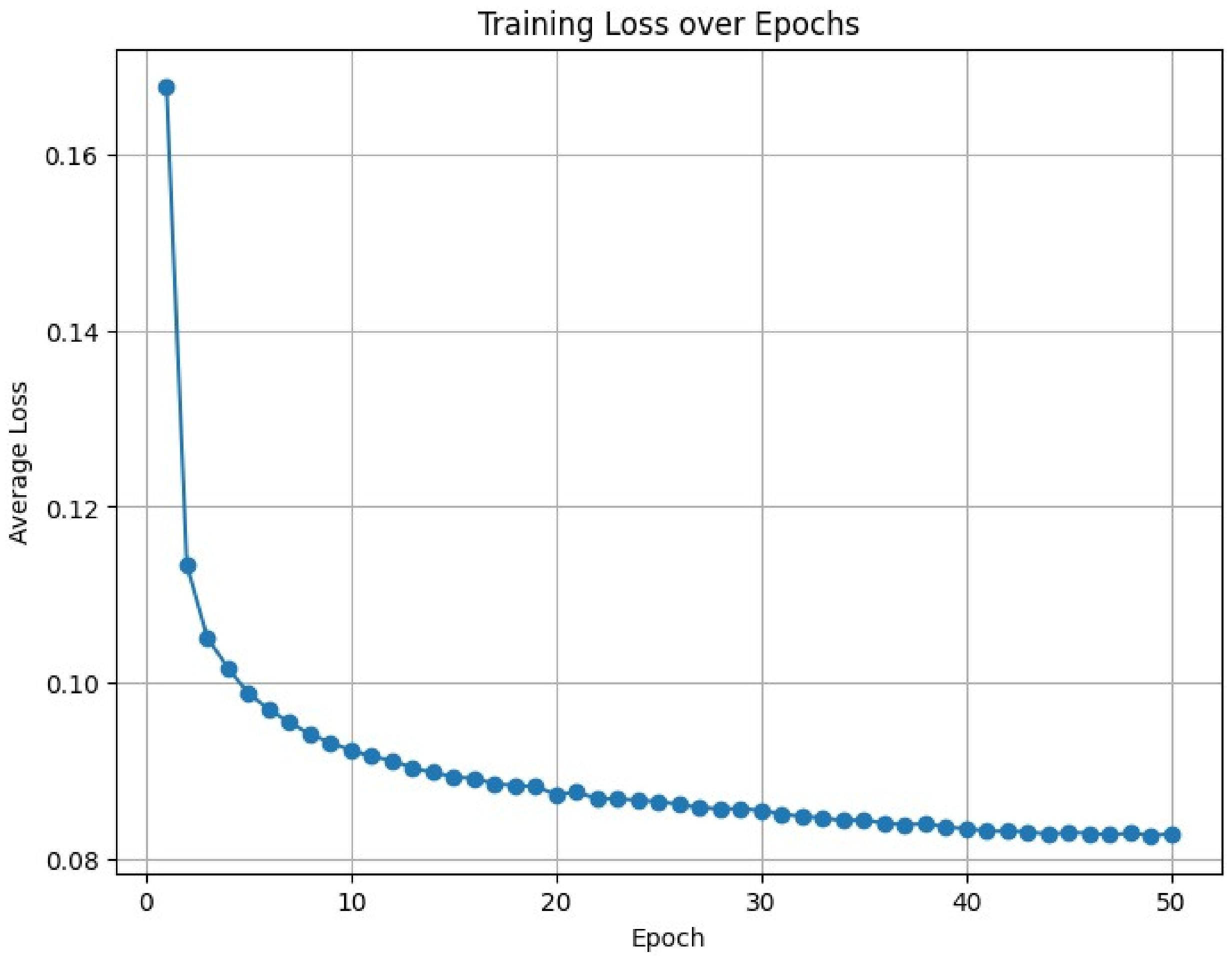
```
Epoch [12/50] Avg Loss: 0.0911
Epoch [13/50] Batch [0] Loss: 0.0913
Epoch [13/50] Batch [100] Loss: 0.0950
Epoch [13/50] Batch [200] Loss: 0.0967
Epoch [13/50] Batch [300] Loss: 0.0908
Epoch [13/50] Avg Loss: 0.0903
Epoch [14/50] Batch [0] Loss: 0.0906
Epoch [14/50] Batch [100] Loss: 0.0851
Epoch [14/50] Batch [200] Loss: 0.0869
Epoch [14/50] Batch [300] Loss: 0.0982
Epoch [14/50] Avg Loss: 0.0898
Epoch [15/50] Batch [0] Loss: 0.0905
Epoch [15/50] Batch [100] Loss: 0.0917
Epoch [15/50] Batch [200] Loss: 0.0964
Epoch [15/50] Batch [300] Loss: 0.0906
Epoch [15/50] Avg Loss: 0.0893
Model saved at epoch 15
Epoch [16/50] Batch [0] Loss: 0.0942
Epoch [16/50] Batch [100] Loss: 0.0916
Epoch [16/50] Batch [200] Loss: 0.0857
Epoch [16/50] Batch [300] Loss: 0.0889
Epoch [16/50] Avg Loss: 0.0891
Epoch [17/50] Batch [0] Loss: 0.0860
Epoch [17/50] Batch [100] Loss: 0.0818
Epoch [17/50] Batch [200] Loss: 0.0915
Epoch [17/50] Batch [300] Loss: 0.0857
Epoch [17/50] Avg Loss: 0.0885
Epoch [18/50] Batch [0] Loss: 0.0859
Epoch [18/50] Batch [100] Loss: 0.0921
Epoch [18/50] Batch [200] Loss: 0.0906
Epoch [18/50] Batch [300] Loss: 0.0858
Epoch [18/50] Avg Loss: 0.0883
Epoch [19/50] Batch [0] Loss: 0.0926
Epoch [19/50] Batch [100] Loss: 0.0884
Epoch [19/50] Batch [200] Loss: 0.0893
Epoch [19/50] Batch [300] Loss: 0.0871
Epoch [19/50] Avg Loss: 0.0882
Epoch [20/50] Batch [0] Loss: 0.0889
Epoch [20/50] Batch [100] Loss: 0.0930
Epoch [20/50] Batch [200] Loss: 0.0890
Epoch [20/50] Batch [300] Loss: 0.0863
Epoch [20/50] Avg Loss: 0.0872
Model saved at epoch 20
Epoch [21/50] Batch [0] Loss: 0.0871
Epoch [21/50] Batch [100] Loss: 0.0832
Epoch [21/50] Batch [200] Loss: 0.0897
Epoch [21/50] Batch [300] Loss: 0.0895
Epoch [21/50] Avg Loss: 0.0876
Epoch [22/50] Batch [0] Loss: 0.0875
```

```
Epoch [22/50] Batch [100] Loss: 0.0836
Epoch [22/50] Batch [200] Loss: 0.0885
Epoch [22/50] Batch [300] Loss: 0.0832
Epoch [22/50] Avg Loss: 0.0868
Epoch [23/50] Batch [0] Loss: 0.0879
Epoch [23/50] Batch [100] Loss: 0.0872
Epoch [23/50] Batch [200] Loss: 0.0922
Epoch [23/50] Batch [300] Loss: 0.0938
Epoch [23/50] Avg Loss: 0.0868
Epoch [24/50] Batch [0] Loss: 0.0838
Epoch [24/50] Batch [100] Loss: 0.0854
Epoch [24/50] Batch [200] Loss: 0.0870
Epoch [24/50] Batch [300] Loss: 0.0950
Epoch [24/50] Avg Loss: 0.0866
Epoch [25/50] Batch [0] Loss: 0.0909
Epoch [25/50] Batch [100] Loss: 0.0872
Epoch [25/50] Batch [200] Loss: 0.0818
Epoch [25/50] Batch [300] Loss: 0.0819
Epoch [25/50] Avg Loss: 0.0865
Model saved at epoch 25
Epoch [26/50] Batch [0] Loss: 0.0839
Epoch [26/50] Batch [100] Loss: 0.0863
Epoch [26/50] Batch [200] Loss: 0.0818
Epoch [26/50] Batch [300] Loss: 0.0863
Epoch [26/50] Avg Loss: 0.0862
Epoch [27/50] Batch [0] Loss: 0.0830
Epoch [27/50] Batch [100] Loss: 0.0836
Epoch [27/50] Batch [200] Loss: 0.0826
Epoch [27/50] Batch [300] Loss: 0.0851
Epoch [27/50] Avg Loss: 0.0858
Epoch [28/50] Batch [0] Loss: 0.0839
Epoch [28/50] Batch [100] Loss: 0.0875
Epoch [28/50] Batch [200] Loss: 0.0830
Epoch [28/50] Batch [300] Loss: 0.0870
Epoch [28/50] Avg Loss: 0.0856
Epoch [29/50] Batch [0] Loss: 0.0817
Epoch [29/50] Batch [100] Loss: 0.0850
Epoch [29/50] Batch [200] Loss: 0.0850
Epoch [29/50] Batch [300] Loss: 0.0824
Epoch [29/50] Avg Loss: 0.0857
Epoch [30/50] Batch [0] Loss: 0.0871
Epoch [30/50] Batch [100] Loss: 0.0947
Epoch [30/50] Batch [200] Loss: 0.0895
Epoch [30/50] Batch [300] Loss: 0.0863
Epoch [30/50] Avg Loss: 0.0855
Model saved at epoch 30
Epoch [31/50] Batch [0] Loss: 0.0850
Epoch [31/50] Batch [100] Loss: 0.0901
Epoch [31/50] Batch [200] Loss: 0.0837
```

```
Epoch [31/50] Batch [300] Loss: 0.0881
Epoch [31/50] Avg Loss: 0.0851
Epoch [32/50] Batch [0] Loss: 0.0857
Epoch [32/50] Batch [100] Loss: 0.0833
Epoch [32/50] Batch [200] Loss: 0.0891
Epoch [32/50] Batch [300] Loss: 0.0831
Epoch [32/50] Avg Loss: 0.0848
Epoch [33/50] Batch [0] Loss: 0.0806
Epoch [33/50] Batch [100] Loss: 0.0847
Epoch [33/50] Batch [200] Loss: 0.0800
Epoch [33/50] Batch [300] Loss: 0.0924
Epoch [33/50] Avg Loss: 0.0846
Epoch [34/50] Batch [0] Loss: 0.0887
Epoch [34/50] Batch [100] Loss: 0.0825
Epoch [34/50] Batch [200] Loss: 0.0849
Epoch [34/50] Batch [300] Loss: 0.0823
Epoch [34/50] Avg Loss: 0.0843
Epoch [35/50] Batch [0] Loss: 0.0819
Epoch [35/50] Batch [100] Loss: 0.0787
Epoch [35/50] Batch [200] Loss: 0.0867
Epoch [35/50] Batch [300] Loss: 0.0778
Epoch [35/50] Avg Loss: 0.0844
Model saved at epoch 35
Epoch [36/50] Batch [0] Loss: 0.0836
Epoch [36/50] Batch [100] Loss: 0.0828
Epoch [36/50] Batch [200] Loss: 0.0816
Epoch [36/50] Batch [300] Loss: 0.0825
Epoch [36/50] Avg Loss: 0.0840
Epoch [37/50] Batch [0] Loss: 0.0917
Epoch [37/50] Batch [100] Loss: 0.0812
Epoch [37/50] Batch [200] Loss: 0.0848
Epoch [37/50] Batch [300] Loss: 0.0830
Epoch [37/50] Avg Loss: 0.0839
Epoch [38/50] Batch [0] Loss: 0.0847
Epoch [38/50] Batch [100] Loss: 0.0831
Epoch [38/50] Batch [200] Loss: 0.0836
Epoch [38/50] Batch [300] Loss: 0.0842
Epoch [38/50] Avg Loss: 0.0840
Epoch [39/50] Batch [0] Loss: 0.0833
Epoch [39/50] Batch [100] Loss: 0.0834
Epoch [39/50] Batch [200] Loss: 0.0806
Epoch [39/50] Batch [300] Loss: 0.0807
Epoch [39/50] Avg Loss: 0.0836
Epoch [40/50] Batch [0] Loss: 0.0774
Epoch [40/50] Batch [100] Loss: 0.0857
Epoch [40/50] Batch [200] Loss: 0.0795
Epoch [40/50] Batch [300] Loss: 0.0806
Epoch [40/50] Avg Loss: 0.0834
Model saved at epoch 40
```

```
Epoch [41/50] Batch [0] Loss: 0.0855
Epoch [41/50] Batch [100] Loss: 0.0859
Epoch [41/50] Batch [200] Loss: 0.0806
Epoch [41/50] Batch [300] Loss: 0.0866
Epoch [41/50] Avg Loss: 0.0831
Epoch [42/50] Batch [0] Loss: 0.0790
Epoch [42/50] Batch [100] Loss: 0.0849
Epoch [42/50] Batch [200] Loss: 0.0858
Epoch [42/50] Batch [300] Loss: 0.0828
Epoch [42/50] Avg Loss: 0.0831
Epoch [43/50] Batch [0] Loss: 0.0866
Epoch [43/50] Batch [100] Loss: 0.0851
Epoch [43/50] Batch [200] Loss: 0.0842
Epoch [43/50] Batch [300] Loss: 0.0797
Epoch [43/50] Avg Loss: 0.0830
Epoch [44/50] Batch [0] Loss: 0.0791
Epoch [44/50] Batch [100] Loss: 0.0823
Epoch [44/50] Batch [200] Loss: 0.0826
Epoch [44/50] Batch [300] Loss: 0.0781
Epoch [44/50] Avg Loss: 0.0827
Epoch [45/50] Batch [0] Loss: 0.0825
Epoch [45/50] Batch [100] Loss: 0.0824
Epoch [45/50] Batch [200] Loss: 0.0807
Epoch [45/50] Batch [300] Loss: 0.0849
Epoch [45/50] Avg Loss: 0.0830
Model saved at epoch 45
Epoch [46/50] Batch [0] Loss: 0.0815
Epoch [46/50] Batch [100] Loss: 0.0828
Epoch [46/50] Batch [200] Loss: 0.0810
Epoch [46/50] Batch [300] Loss: 0.0815
Epoch [46/50] Avg Loss: 0.0829
Epoch [47/50] Batch [0] Loss: 0.0791
Epoch [47/50] Batch [100] Loss: 0.0825
Epoch [47/50] Batch [200] Loss: 0.0830
Epoch [47/50] Batch [300] Loss: 0.0788
Epoch [47/50] Avg Loss: 0.0827
Epoch [48/50] Batch [0] Loss: 0.0784
Epoch [48/50] Batch [100] Loss: 0.0830
Epoch [48/50] Batch [200] Loss: 0.0777
Epoch [48/50] Batch [300] Loss: 0.0842
Epoch [48/50] Avg Loss: 0.0829
Epoch [49/50] Batch [0] Loss: 0.0848
Epoch [49/50] Batch [100] Loss: 0.0826
Epoch [49/50] Batch [200] Loss: 0.0856
Epoch [49/50] Batch [300] Loss: 0.0794
Epoch [49/50] Avg Loss: 0.0826
Epoch [50/50] Batch [0] Loss: 0.0815
Epoch [50/50] Batch [100] Loss: 0.0775
Epoch [50/50] Batch [200] Loss: 0.0809
```

```
Epoch [50/50] Batch [300] Loss: 0.0815
Epoch [50/50] Avg Loss: 0.0828
Model saved at epoch 50
Training complete!
```




```
def visualize_images_with_titles(images: torch.Tensor, column_names: list[str]):
    num_images, num_columns = images.shape[0], len(column_names)
    fig, axes = plt.subplots(num_images, num_columns, figsize=(num_columns,num_images))

    for i, axr in enumerate(axes):
        for j, axc in enumerate(axr):
            img = images[i,j].cpu().numpy()

            axc.imshow(img, cmap='gray')
            axc.axis('off')

            if i == 0:
                axc.set_title(column_names[j])
```

```
    plt.tight_layout(pad=1)
    plt.show()

unet = TimeConditionalUNet(in_channels=1, num_classes=1,
num_hiddens=64).to(device)
model = RF(unet=unet, num_ts=300).to(device)

epoch_list = [1] + [i for i in range(5, epochs+1, 5)]
sampled_images_list = []
titles = []

for i in epoch_list:

model.load_state_dict(torch.load(f"./rf_models/model_epoch_{i}.pth"))
    sampled = model.sample(img_wh=(28, 28), seed=5)
    sampled_images_list.append(sampled)
    titles.append(f'Epoch {i}')

all_samples = torch.hstack(sampled_images_list)

axes = visualize_images_with_titles(all_samples, titles)
```