

▼ Project 5B: Flow Matching from Scratch!

▼ Setup environment

```
# We recommend using these utils.
# https://google.github.io/mediapy/mediapy.html
# https://einops.rocks/
!pip install mediapy einops --quiet
```

1.6/1.6 MB 31.2 MB/s eta 0:00:00

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

Neural Network Resources

In this part, you will build and train a [UNet](#), which is more complex than the MLP you implemented in the NeRF project. We provide all class definitions you may need (but feel free to add or modify them as necessary).

Instead of asking ChatGPT to write everything for you, please consult the following resources when you get stuck — they will help you understand how and why things work under the hood.

- PyTorch Documentation – [Conv2d](#), [ConvTranspose2d](#), and [AvgPool2d](#).
- PyTorch Documentation – [torchvision.datasets.MNIST](#), the dataset we gonna use, and [torch.utils.data.DataLoader](#), the off-the-shell dataloader we can directly use.
- PyTorch [tutorial](#) on how to train a classifier on CIFAR10 dataset. The structure of your training code will be very similar to this one.

Part 1: Training a Single-step Denoising UNet

▼ Part 1.1: Implementing the UNet

▼ Implementing Simple and Composed Ops

```
class Conv(nn.Module):
    def __init__(self, in_channels: int, out_channels: int):
        super().__init__()
        # ===== your code here! =====
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size = 3, stride = 1, padding = 1),
            nn.BatchNorm2d(out_channels),
            nn.GELU()
        )
        # ===== end of code =====

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # ===== your code here! =====
        out = self.conv(x)
        return out

        # ===== end of code =====
        raise NotImplementedError()

class DownConv(nn.Module):
    def __init__(self, in_channels: int, out_channels: int):
        super().__init__()
        # ===== your code here! =====
        self.DownConv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size = 3, stride = 2, padding = 1),
            nn.BatchNorm2d(out_channels),
```

```
        nn.GELU()
    )

# ===== end of code =====

def forward(self, x: torch.Tensor) -> torch.Tensor:
    # ===== your code here! =====
    out = self.DownConv(x)
    return out

    # ===== end of code =====
    raise NotImplementedError()

class UpConv(nn.Module):
    def __init__(self, in_channels: int, out_channels: int):
        super().__init__()
        # ===== your code here! =====
        self.UpConv = nn.Sequential(
            nn.ConvTranspose2d(in_channels, out_channels, kernel_size = 4, stride = 2, padding = 1),
            nn.BatchNorm2d(out_channels),
            nn.GELU()
        )

    # ===== end of code =====

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # ===== your code here! =====
        out = self.UpConv(x)
        return out

    # ===== end of code =====
    raise NotImplementedError()

class Flatten(nn.Module):
    def __init__(self):
        super().__init__()
        # ===== your code here! =====
        self.Flatten = nn.Sequential(
            nn.AvgPool2d(7),
            nn.GELU()
        )

    # ===== end of code =====

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # ===== your code here! =====
        out = self.Flatten(x)
        return out

    # ===== end of code =====
    raise NotImplementedError()

class Unflatten(nn.Module):
    def __init__(self, in_channels: int):
        super().__init__()
        # ===== your code here! =====
        self.Unflatten = nn.Sequential(
            nn.ConvTranspose2d(in_channels, in_channels, kernel_size = 7, stride = 7, padding = 0),
            nn.BatchNorm2d(in_channels),
            nn.GELU()
        )

    # ===== end of code =====

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # ===== your code here! =====
        out = self.Unflatten(x)
        return out

    # ===== end of code =====
    raise NotImplementedError()

class ConvBlock(nn.Module):
    def __init__(self, in_channels: int, out_channels: int):
        super().__init__()
        # ===== your code here! =====
        self.ConvBlock = nn.Sequential(
            Conv(in_channels, out_channels),
```

```

        Conv(out_channels, out_channels)
    )

# ===== end of code =====

def forward(self, x: torch.Tensor) -> torch.Tensor:
    # ===== your code here! =====
    out = self.ConvBlock(x)
    return out

# ===== end of code =====
raise NotImplementedError()

class DownBlock(nn.Module):
    def __init__(self, in_channels: int, out_channels: int):
        super().__init__()
        # ===== your code here! =====
        self.DownBlock = nn.Sequential(
            DownConv(in_channels, out_channels),
            ConvBlock(out_channels, out_channels),
        )

# ===== end of code =====

def forward(self, x: torch.Tensor) -> torch.Tensor:
    # ===== your code here! =====
    out = self.DownBlock(x)
    return out

# ===== end of code =====
raise NotImplementedError()

class UpBlock(nn.Module):
    def __init__(self, in_channels: int, out_channels: int):
        super().__init__()
        # ===== your code here! =====
        self.UpBlock = nn.Sequential(
            UpConv(in_channels, out_channels),
            ConvBlock(out_channels, out_channels),
        )

# ===== end of code =====

def forward(self, x: torch.Tensor) -> torch.Tensor:
    # ===== your code here! =====
    out = self.UpBlock(x)
    return out

# ===== end of code =====
raise NotImplementedError()

```

▼ Implementing Unconditional UNet

```

class UnconditionalUNet(nn.Module):
    def __init__(
        self,
        in_channels: int,
        num_hiddens: int,
    ):
        super().__init__()
        # ===== your code here! =====
        D = num_hiddens
        self.conv1 = ConvBlock(in_channels, D)
        self.down1 = DownBlock(D, D)
        self.down2 = DownBlock(D, 2 * D)
        self.flatten = Flatten()

        self.unflatten = Unflatten(2 * D)

        self.up1 = UpBlock(4*D, D)

        self.up2 = UpBlock(2*D, D)

        self.conv2 = ConvBlock(2*D, D)
        self.conv3 = nn.Conv2d(D, 1, kernel_size = 3, stride = 1, padding = 1)

# ===== end of code =====

```

```

def forward(self, x: torch.Tensor) -> torch.Tensor:
    assert x.shape[-2:] == (28, 28), "Expect input shape to be (28, 28)."
    # ===== your code here! =====
    x0 = self.conv1(x)
    x1 = self.down1(x0)
    x2 = self.down2(x1)

    b = self.flatten(x2)
    b = self.unflatten(b)

    x3 = torch.cat([x2, b], dim = 1)
    x4 = self.up1(x3)
    x5 = torch.cat([x1, x4], dim = 1)
    x6 = self.up2(x5)
    x7 = torch.cat([x0, x6], dim = 1)
    x8 = self.conv2(x7)
    x9 = self.conv3(x8)
    return x9

# ===== end of code =====
raise NotImplementedError()

```

▼ Part 1.2: Using the UNet to Train a Denoiser

```

# Visualize images at different noisy level
# ===== your code here! =====
dataset = MNIST(root="data", train=True, download=True, transform=ToTensor())

x, _ = dataset[0]      # x shape = (1, 28, 28)
x = x.to(torch.float32)

sigmas = [0.0, 0.2, 0.4, 0.5, 0.6, 0.8, 1.0]

plt.figure(figsize=(14, 3))

for i, sigma in enumerate(sigmas):
    noise = torch.randn_like(x)
    z = torch.clamp(x + sigma * noise, 0.0, 1.0)

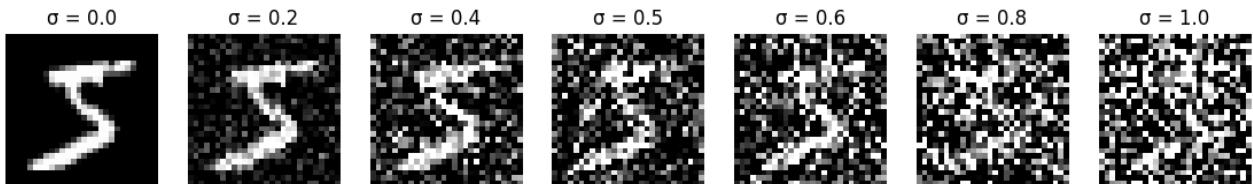
    plt.subplot(1, len(sigmas), i + 1)
    plt.imshow(z.squeeze().numpy(), cmap="gray")
    plt.title(f"σ = {sigma}")
    plt.axis("off")

plt.show()

# ===== end of code =====

```

100%|██████████| 9.91M/9.91M [00:00<00:00, 18.0MB/s]
100%|██████████| 28.9k/28.9k [00:00<00:00, 480kB/s]
100%|██████████| 1.65M/1.65M [00:00<00:00, 4.47MB/s]
100%|██████████| 4.54k/4.54k [00:00<00:00, 8.33MB/s]



▼ Part 1.2.1: Training

For this part, we provide some structure code for training. It is very basic, so feel free to change them or add your code. In later section we won't provide any training or visualization structure code.

```

device = torch.device('cuda')

# Set your hyperparameters
# ===== your code here! =====
batch_size = 256
learning_rate = 1e-4
noise_level = 0.5
hidden_dim = 128

```

```
num_epochs = 5
# ===== end of code =====
```

```
# Define your datasets and dataloaders
# ===== your code here! =====
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=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
# ===== end of code =====
```

```
# Define your model, optimizer, and loss
# ===== your code here! =====
model = UnconditionalUNet(in_channels=1, num_hiddens=hidden_dim).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
criterion = nn.MSELoss()
# ===== end of code =====
```

```
# The training loops
train_losses = []

def visualize_epoch(epoch_idx):
    model.eval()
    with torch.no_grad():
        test_images, _ = next(iter(test_loader))
        test_images = test_images.to(device)

        # Add Gaussian noise with σ = 0.5
        noise = torch.randn_like(test_images)
        noisy_test = torch.clamp(test_images + noise_level * noise, 0.0, 1.0)

        # Denoise using UNet
        denoised = model(noisy_test)

        # Plot clean | noisy | denoised
        plt.figure(figsize=(10, 4))

        for i in range(6):  # show 6 samples
            # CLEAN
            plt.subplot(3, 6, i + 1)
            plt.imshow(test_images[i].squeeze().cpu(), cmap="gray")
            plt.axis("off")
            if i == 0:
                plt.ylabel("Input")

            # NOISY
            plt.subplot(3, 6, 6 + i + 1)
            plt.imshow(noisy_test[i].squeeze().cpu(), cmap="gray")
            plt.axis("off")
            if i == 0:
                plt.ylabel("Noisy (σ=0.5)")

            # DENOISED OUTPUT
            plt.subplot(3, 6, 12 + i + 1)
            plt.imshow(denoised[i].squeeze().cpu(), cmap="gray")
            plt.axis("off")
            if i == 0:
                plt.ylabel("Output")

        plt.suptitle(f"Denoising Results After Epoch {epoch_idx}")
        plt.show()

for epoch in range(num_epochs):
    model.train()
    for i, (images, _) in enumerate(tqdm(train_loader)):
        images = images.to(device)
        noise = torch.randn_like(images)
        noisy_images = torch.clamp(images + noise_level * noise, 0.0, 1.0)

        # ===== your code here! =====
        outputs = model(noisy_images)

        # ===== end of code =====
        loss = criterion(outputs, images)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

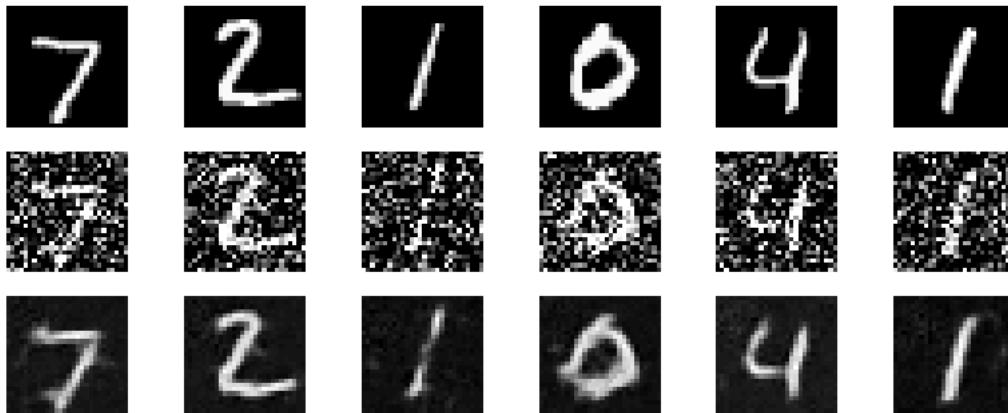
```

    train_losses.append(loss.item())
    print(f"Epoch {epoch+1} finished, loss = {loss.item():.4f}")
    if (epoch + 1) in [1, num_epochs]:
        visualize_epoch(epoch + 1)

100%|██████████| 235/235 [01:31<00:00,  2.56it/s]
Epoch 1 finished, loss = 0.0124

```

Denoising Results After Epoch 1

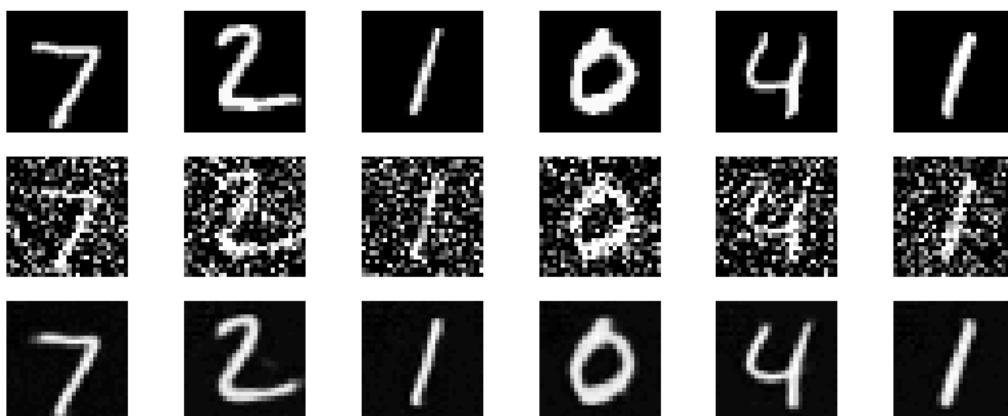


```

100%|██████████| 235/235 [01:38<00:00,  2.39it/s]
Epoch 2 finished, loss = 0.0107
100%|██████████| 235/235 [01:38<00:00,  2.40it/s]
Epoch 3 finished, loss = 0.0095
100%|██████████| 235/235 [01:37<00:00,  2.40it/s]
Epoch 4 finished, loss = 0.0089
100%|██████████| 235/235 [01:37<00:00,  2.40it/s]
Epoch 5 finished, loss = 0.0085

```

Denoising Results After Epoch 5

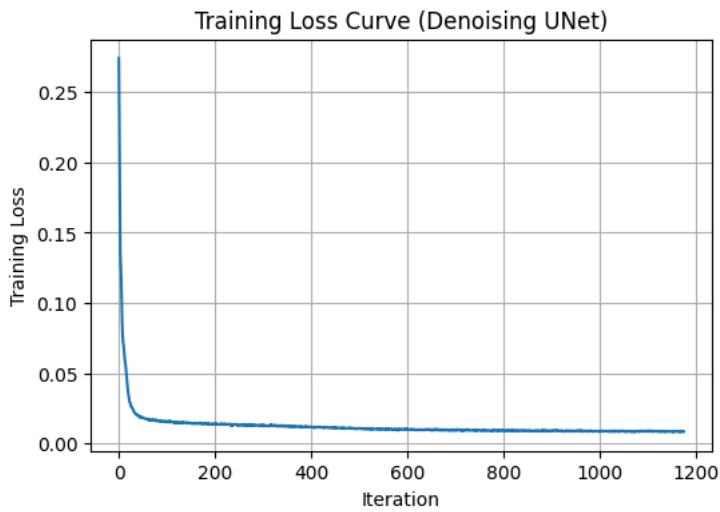


```

# Visualize your training curve
# ===== your code here! =====
plt.figure(figsize=(6,4))
plt.plot(train_losses)
plt.xlabel("Iteration")
plt.ylabel("Training Loss")
plt.title("Training Loss Curve (Denoising UNet)")
plt.grid(True)
plt.show()

# ===== end of code =====

```



▼ Part 1.2.2: Out-of-Distribution Testing

```
# Visualize OOD testing
# ===== your code here! =====
def visualize_ood_noise_levels(model, test_loader, sigma_levels):
    model.eval()

    # get a single test image
    images, _ = next(iter(test_loader))
    img = images[0:1].to(device)

    num_sigmas = len(sigma_levels)

    # TWO ROWS: Row 1 = Noisy images, Row 2 = Denoised results
    fig, axs = plt.subplots(2, num_sigmas, figsize=(3 * num_sigmas, 6))

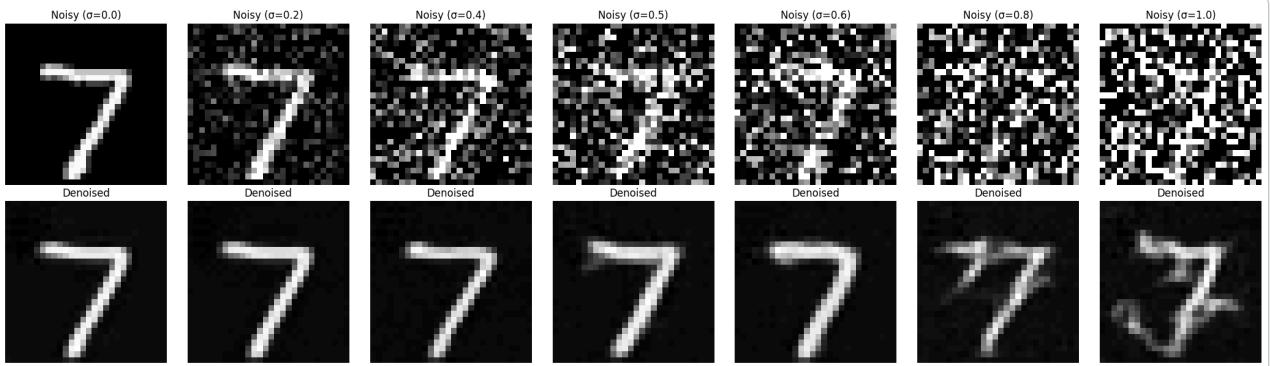
    for i, sigma in enumerate(sigma_levels):
        # generate noise
        noise = sigma * torch.randn_like(img)
        noisy = torch.clamp(img + noise, 0.0, 1.0)

        # denoise
        with torch.no_grad():
            denoised = model(noisy)

        # row 1 = noisy
        axs[0, i].imshow(noisy[0, 0].cpu(), cmap="gray")
        axs[0, i].set_title(f"Noisy (σ={sigma})")
        axs[0, i].axis("off")

        # row 2 = denoised
        axs[1, i].imshow(denoised[0, 0].cpu(), cmap="gray")
        axs[1, i].set_title("Denoised")
        axs[1, i].axis("off")

    plt.tight_layout()
    plt.show()
# ===== end of code =====
sigma_list = [0.0, 0.2, 0.4, 0.5, 0.6, 0.8, 1.0]
visualize_ood_noise_levels(model, test_loader, sigma_list)
```



Part 1.2.3 Denoising Pure Noise

```
def visualize_pure_noise_denoising(model, test_loader, epoch, num_samples=6):
    model.eval()

    # create pure noise input of same shape as first batch
    images, _ = next(iter(test_loader))
    images = images[:num_samples].to(device)
    noise = torch.randn_like(images)

    # denoise
    with torch.no_grad():
        outputs = model(noise)

    # TWO ROWS, num_samples COLUMNS
    fig, axs = plt.subplots(2, num_samples, figsize=(3 * num_samples, 6))

    for i in range(num_samples):

        # ---- Row 1: Noise ----
        axs[0, i].imshow(noise[i, 0].cpu(), cmap="gray")
        axs[0, i].set_title(f"Noise {i+1}")
        axs[0, i].axis("off")

        # ---- Row 2: Output ----
        axs[1, i].imshow(outputs[i, 0].cpu(), cmap="gray")
        axs[1, i].set_title(f"Denoised (Epoch {epoch})")
        axs[1, i].axis("off")

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

```
# Feel free to use code from part 1.2.1
# as they should be very similar
# ===== your code here! =====
pure_noise_model = UnconditionalUNet(in_channels=1, num_hiddens=128).to(device)

criterion = nn.MSELoss()
optimizer = torch.optim.Adam(pure_noise_model.parameters(), lr=1e-4)

num_epochs = 5
train_losses = []

for epoch in range(num_epochs):
    pure_noise_model.train()

    for images, _ in tqdm(train_loader):
        images = images.to(device)
        noise = torch.randn_like(images)

        outputs = pure_noise_model(noise)
        loss = criterion(outputs, images)

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

```

        train_losses.append(loss.item())

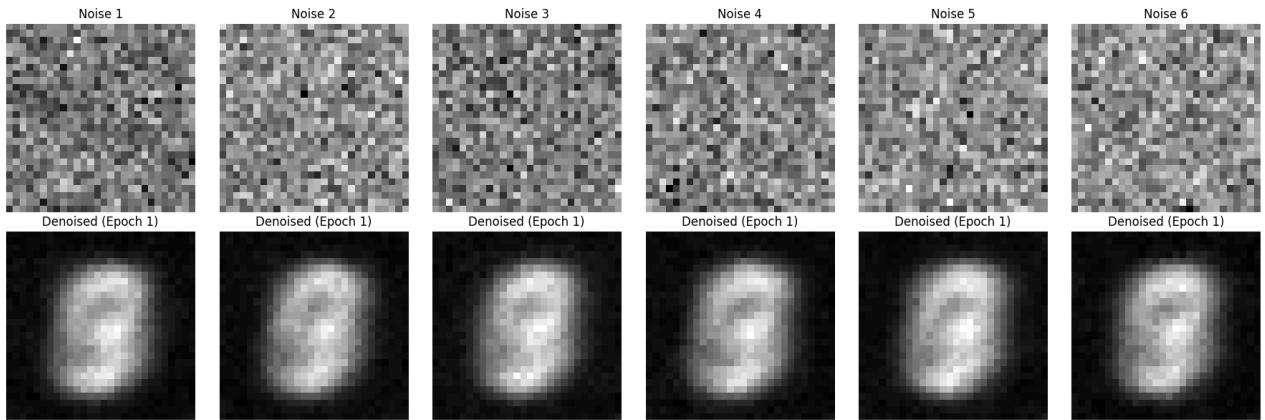
        print(f"Epoch {epoch+1} → loss: {loss.item():.4f}")

    # visualize after epoch 1 and 5
    if epoch == 0 or epoch == num_epochs - 1:
        visualize_pure_noise_denoising(pure_noise_model, test_loader, epoch+1)

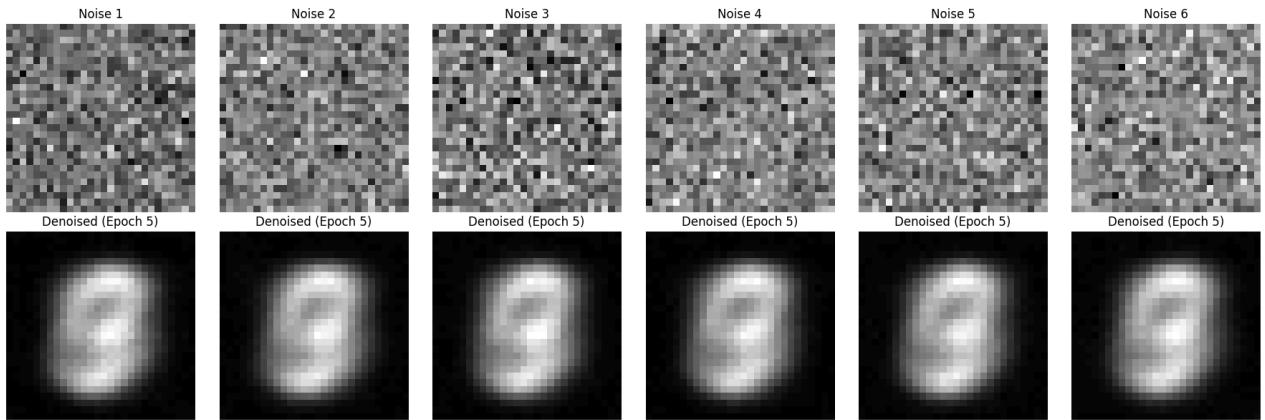
# ===== end of code =====

```

100%|██████████| 235/235 [01:38<00:00, 2.40it/s]
Epoch 1 → loss: 0.0672



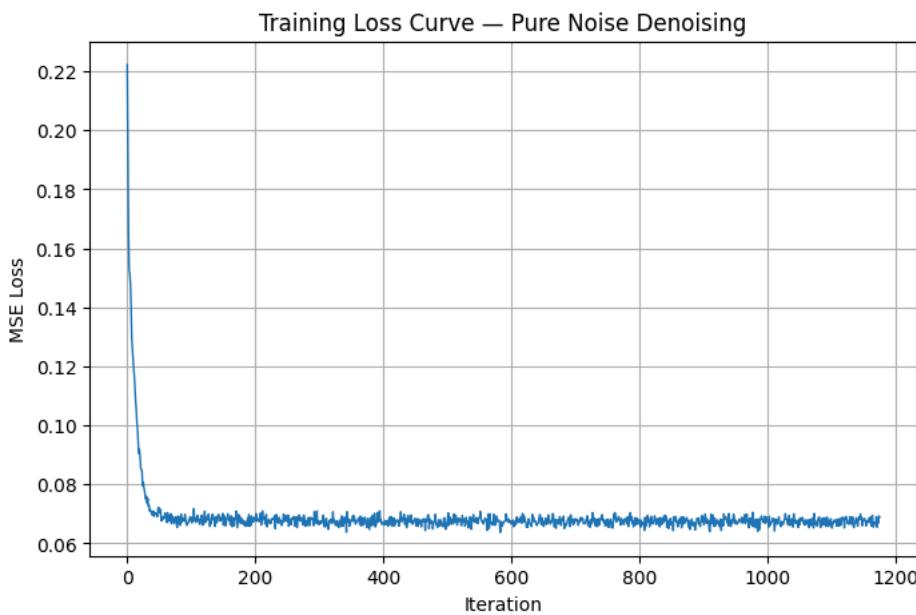
100%|██████████| 235/235 [01:37<00:00, 2.40it/s]
Epoch 2 → loss: 0.0696
100%|██████████| 235/235 [01:38<00:00, 2.40it/s]
Epoch 3 → loss: 0.0661
100%|██████████| 235/235 [01:37<00:00, 2.40it/s]
Epoch 4 → loss: 0.0701
100%|██████████| 235/235 [01:37<00:00, 2.40it/s]
Epoch 5 → loss: 0.0689



```

plt.figure(figsize=(8,5))
plt.plot(train_losses, linewidth=1)
plt.title("Training Loss Curve – Pure Noise Denoising")
plt.xlabel("Iteration")
plt.ylabel("MSE Loss")
plt.grid(True)
plt.show()

```



Part 2: Flow Matching

▼ Part 2.1: Implementing a Time-conditioned UNet

```

class FCBLOCK(nn.Module):
    def __init__(self, in_channels: int, out_channels: int):
        super().__init__()
        # ===== your code here! =====
        self.fcblock = nn.Sequential(
            nn.Linear(in_channels, out_channels),
            nn.GELU(),
            nn.Linear(out_channels, out_channels)
        )
        # ===== end of code =====

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # ===== your code here! =====
        if x.dim() == 1:
            x = x.unsqueeze(1)
        out = self.fcblock(x)
        out = out.unsqueeze(-1).unsqueeze(-1)
        return out

        # ===== end of code =====
        raise NotImplementedError()

class TimeConditionalUNet(nn.Module):
    def __init__(
        self,
        in_channels: int,
        num_classes: int,
        num_hiddens: int,
    ):
        super().__init__()
        # ===== your code here! =====
        D = num_hiddens
        self.conv1 = ConvBlock(in_channels, D)
        self.down1 = DownBlock(D, D)
        self.down2 = DownBlock(D, 2 * D)
        self.flatten = Flatten()

        self.unflatten = Unflatten(2 * D)

        self.up1 = UpBlock(4*D, D)
        self.up2 = UpBlock(2*D, D)

        self.conv2 = ConvBlock(2*D, D)
        self.conv3 = nn.Conv2d(D, 1, kernel_size = 3, stride = 1, padding = 1)

```

```

    self.time1 = FCBLOCK(1, 2*D)
    self.time2 = FCBLOCK(1, D)

    # ===== end of code =====

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)."
    # ===== your code here! =====
    x0 = self.conv1(x)           # (D, 28x28)
    x1 = self.down1(x0)          # (D, 14x14)
    x2 = self.down2(x1)          # (2D, 7x7)

    b = self.flatten(x2)         # (2D, 1, 1)
    b = self.unflatten(b)        # (2D, 7, 7)

    t1 = self.time1(t)
    b = b + t1
    x3 = torch.cat([x2, b], dim = 1)
    x4 = self.up1(x3)
    t2 = self.time2(t)
    x4 = t2 + x4
    x5 = torch.cat([x1, x4], dim = 1)
    x6 = self.up2(x5)
    x7 = torch.cat([x0, x6], dim = 1)
    x8 = self.conv2(x7)
    x9 = self.conv3(x8)
    return x9

    # ===== end of code =====
    raise NotImplementedError()

```

▼ Implementing the Forward and Reverse Process for Time-conditioned Denoising

```

def time_fm_forward(
    unet: TimeConditionalUNet,
    x_1: torch.Tensor,
    num_ts: int,
) -> torch.Tensor:
    """Algorithm 1

    Args:
        unet: TimeConditionalUNet
        x_1: (N, C, H, W) input tensor.
        num_ts: int, number of timesteps.

    Returns:
        (,) loss.
    """
    unet.train()

    device = x_1.device
    N = x_1.shape[0]
    t_idx = torch.randint(0, num_ts, (N,), device=device)
    t = t_idx / (num_ts - 1)

    x_0 = torch.randn_like(x_1)
    t_broadcast = t.view(N, 1, 1, 1)
    x_t = (1 - t_broadcast) * x_0 + t_broadcast * x_1
    u_gt = x_1 - x_0

    u_pred = unet(x_t, t)
    loss = torch.mean((u_pred - u_gt) ** 2)

    return loss
    # ===== end of code =====
    raise NotImplementedError()

```

```

@torch.inference_mode()
def time_fm_sample(
    unet: TimeConditionalUNet,
    img_wh: tuple[int, int],
    num_ts: int,
    seed: int = 0,
    batch_size: int = 25,
) -> torch.Tensor:

    unet.eval()
    torch.manual_seed(seed)

    H, W = img_wh
    device = next(unet.parameters()).device

    x = torch.randn(batch_size, 1, H, W, device=device)

    dt = 1.0 / num_ts
    for k in range(num_ts):
        t_val = k / (num_ts - 1)
        t = torch.full((batch_size,), t_val, device=device)
        u = unet(x, t)
        x = x + dt * u

    return x

```

```

class TimeConditionalFM(nn.Module):
    def __init__(
        self,
        unet: TimeConditionalUNet,
        num_ts: int = 50,
        img_hw: tuple[int, int] = (28, 28),
    ):
        super().__init__()

        self.unet = unet
        self.num_ts = num_ts
        self.img_hw = img_hw

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

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

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

```

Start coding or generate with AI.

Part 2.2: Training the Time-conditioned UNet

```

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=64, shuffle=False)

```

```

# Feel free to use code from part 1.2.1
# as they should be very similar
# ===== your code here! =====
num_epochs = 10

```

```

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

model = TimeConditionalFM(
    unet=unet,
    num_ts=50
).to(device)

optimizer = torch.optim.Adam(model.parameters(), lr=1e-2)

scheduler = torch.optim.lr_scheduler.ExponentialLR(
    optimizer,
    gamma=0.1 ** (1 / num_epochs)
)

train_losses = []

for epoch in range(num_epochs):
    model.train()

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

        loss = model(images)

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

        train_losses.append(loss.item())

    scheduler.step()
    print(f"Epoch {epoch+1}, loss = {loss.item():.4f}")
# ===== end of code =====

100%|██████████| 938/938 [00:42<00:00, 21.97it/s]
Epoch 1, loss = 0.1574
100%|██████████| 938/938 [00:41<00:00, 22.47it/s]
Epoch 2, loss = 0.1433
100%|██████████| 938/938 [00:41<00:00, 22.47it/s]
Epoch 3, loss = 0.1781
100%|██████████| 938/938 [00:41<00:00, 22.47it/s]
Epoch 4, loss = 0.1339
100%|██████████| 938/938 [00:41<00:00, 22.54it/s]
Epoch 5, loss = 0.1061
100%|██████████| 938/938 [00:41<00:00, 22.40it/s]
Epoch 6, loss = 0.0888
100%|██████████| 938/938 [00:41<00:00, 22.47it/s]
Epoch 7, loss = 0.0876
100%|██████████| 938/938 [00:41<00:00, 22.41it/s]
Epoch 8, loss = 0.0865
100%|██████████| 938/938 [00:41<00:00, 22.50it/s]
Epoch 9, loss = 0.0862
100%|██████████| 938/938 [00:41<00:00, 22.44it/s]Epoch 10, loss = 0.1184

```

```

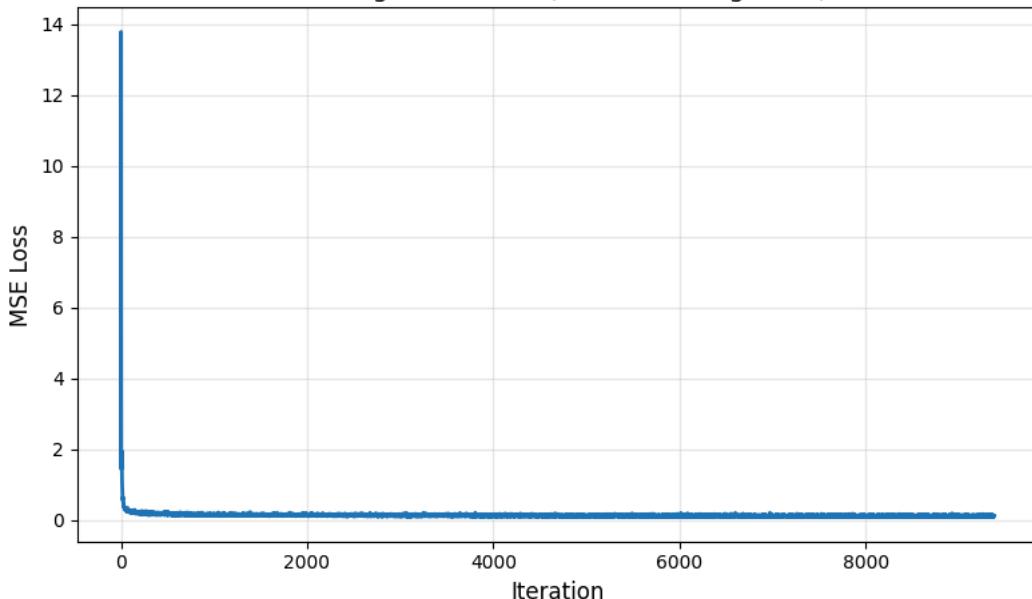
import matplotlib.pyplot as plt
import numpy as np

# Convert to numpy array for easier manipulation
loss_array = np.array(train_losses)

plt.figure(figsize=(8, 5))
plt.plot(loss_array, linewidth=2)
plt.title("Training Loss Curve (Flow Matching UNet)", fontsize=14)
plt.xlabel("Iteration", fontsize=12)
plt.ylabel("MSE Loss", fontsize
           =12)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

```

Training Loss Curve (Flow Matching UNet)



- Part 2.3: Sampling from the Time-conditioned UNet

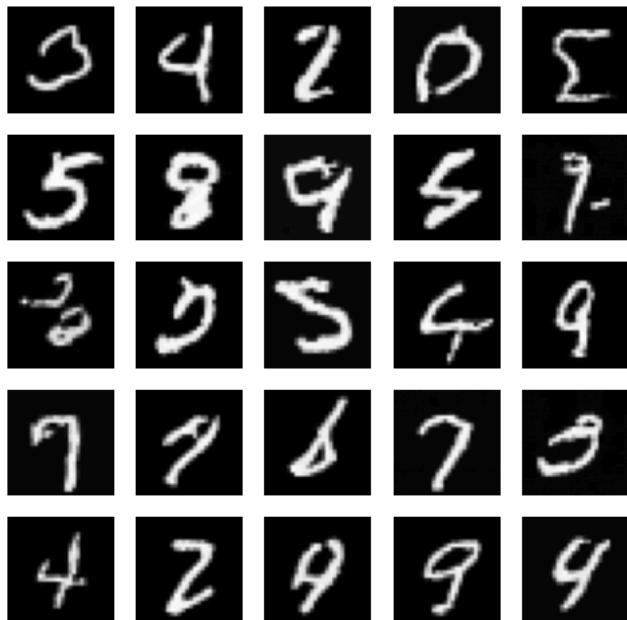
```

samples = time_fm_sample(unet, (28, 28), num_ts=50)

# visualize
fig, axs = plt.subplots(5, 5, figsize=(6,6))
for i, ax in enumerate(axs.flatten()):
    ax.imshow(samples[i,0].cpu(), cmap="gray")
    ax.axis("off")
plt.suptitle("Samples after training epoch 10")
plt.show()

```

Samples after training epoch 10



- Part 2.4: Implementing a Class-conditioned UNet

```

class ClassConditionalUNet(nn.Module):
    def __init__(_
        self,
        in_channels: int,
        num_classes: int,
        num_hiddens: int,

```

```

):_
    super().__init__()
    # ===== your code here! =====
    D = num_hiddens

    # ---- Encoder ----
    self.conv1 = ConvBlock(in_channels, D)
    self.down1 = DownBlock(D, D)
    self.down2 = DownBlock(D, 2 * D)
    self.flatten = Flatten()

    # ---- Bottleneck ----
    self.unflatten = Unflatten(2 * D)

    # ---- Decoder ----
    self.up1 = UpBlock(4 * D, D)
    self.up2 = UpBlock(2 * D, D)

    self.conv2 = ConvBlock(2 * D, D)
    self.conv3 = nn.Conv2d(D, 1, kernel_size=3, stride=1, padding=1)

    # ---- Time Conditioning (same as before) ----
    self.time1 = FCBLOCK(1, 2 * D)
    self.time2 = FCBLOCK(1, D)

    # ---- Class Conditioning (NEW) ----
    self.class1 = FCBLOCK(num_classes, 2 * D)
    self.class2 = FCBLOCK(num_classes, D)

    # ===== end of code =====

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)."
    # ===== your code here! =====
    N = x.size(0)

    # ---- Convert class indices to one-hot ----
    c_onehot = torch.nn.functional.one_hot(c, num_classes=10).float().to(x.device)

    # ---- Apply mask (drop class conditioning 10% of time) ----
    if mask is not None:
        # mask shape: (N,), 1 = keep class, 0 = drop class
        mask = mask.view(N, 1)           # reshape for broadcasting
        c_onehot = c_onehot * mask      # zero out row if mask = 0

    x0 = self.conv1(x)
    x1 = self.down1(x0)
    x2 = self.down2(x1)

    b = self.flatten(x2)
    b = self.unflatten(b)

    t1 = self.time1(t)             # shape → (N, 2D, 1, 1)
    t2 = self.time2(t)             # shape → (N, D, 1, 1)

    c1 = self.class1(c_onehot)    # shape → (N, 2D, 1, 1)
    c2 = self.class2(c_onehot)    # shape → (N, D, 1, 1)

    # Inject conditioning after unflatten (FiLM mod)
    b = c1 * b + t1

    # Start decoder
    x3 = torch.cat([x2, b], dim=1)
    x4 = self.up1(x3)

```

```
# Inject conditioning before second upblock
x4 = c2 * x4 + t2

x5 = torch.cat([x1, x4], dim=1)
x6 = self.up2(x5)

x7 = torch.cat([x0, x6], dim=1)
x8 = self.conv2(x7)
x9 = self.conv3(x8)

return x9
# ===== end of code =====
raise NotImplementedError()
```

```
def class_fm_forward(
    unet: ClassConditionalUNet,
    x_1: torch.Tensor,
    c: torch.Tensor,
    p_uncond: float,
    num_ts: int,
) -> torch.Tensor:
    """Algorithm 3

Args:
    unet: ClassConditionalUNet
    x_1: (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:
    (,) loss.
"""
    unet.train()
    # ===== your code here! =====

    device = x_1.device
    N = x_1.size(0)
    t_idx = torch.randint(0, num_ts, (N,), device=device)
    t = t_idx / (num_ts - 1)      # normalize to [0,1]

    x_0 = torch.randn_like(x_1)

    t_broadcast = t.view(N, 1, 1, 1)
    x_t = (1 - t_broadcast) * x_0 + t_broadcast * x_1

    u_gt = x_1 - x_0          # (N, C, H, W)

    mask = (torch.rand(N, device=device) > p_uncond).float()

    u_pred = unet(x_t, c, t, mask)

    loss = torch.mean((u_pred - u_gt) ** 2)

    return loss

# ===== end of code =====
raise NotImplementedError()
```

```
@torch.inference_mode()
def class_fm_sample(
    unet: ClassConditionalUNet,
    c: torch.Tensor,
    img_wh: tuple[int, int],
    num_ts: int,
    guidance_scale: float = 5.0,
    seed: int = 0,
):
    """
    Batch size is implicitly c.shape[0].
    """
    unet.eval()
    torch.manual_seed(seed)

    # batch_size already determined from c
    N = c.shape[0]
    H, W = img_wh
    device = next(unet.parameters()).device
```

```

x = torch.randn(N, 1, H, W, device=device)

caches = torch.zeros(N, num_ts, 1, H, W, device=device)

dt = 1.0 / num_ts

for k in range(num_ts):
    t_val = k / (num_ts - 1)
    t = torch.full((N,), t_val, device=device)

    u_uncond = unet(x, c, t, mask=torch.zeros(N, device=device))

    u_cond = unet(x, c, t, mask=torch.ones(N, device=device))

    u = u_uncond + guidance_scale * (u_cond - u_uncond)
    x = x + dt * u

    caches[:, k] = x

return x, caches

```

```

class ClassConditionalFM(nn.Module):
    def __init__(
        self,
        unet: ClassConditionalUNet,
        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, c: torch.Tensor) -> torch.Tensor:
        """
        Args:
            x: (N, C, H, W) input tensor.
            c: (N,) int64 condition tensor.

        Returns:
            (,) loss.
        """
        return class_fm_forward(
            self.unet, 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 class_fm_sample(
            self.unet, c, img_wh, self.num_ts, guidance_scale, seed
        )

```

▼ Part 2.5 Training the Class-conditioned UNet

```

# Feel free to use code from part 1.2.1
# as they should be very similar
# ===== your code here! =====
num_epochs = 10

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

model = ClassConditionalFM(
    unet=unet,
    num_ts=50,
    p_uncond=0.1
).to(device)

optimizer = torch.optim.Adam(model.parameters(), lr=1e-2)

```

```
train_losses = []

for epoch in range(num_epochs):
    model.train()

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

        loss = model(images, labels)

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

    train_losses.append(loss.item())

    print(f"Epoch {epoch+1}, loss = {loss.item():.4f}")

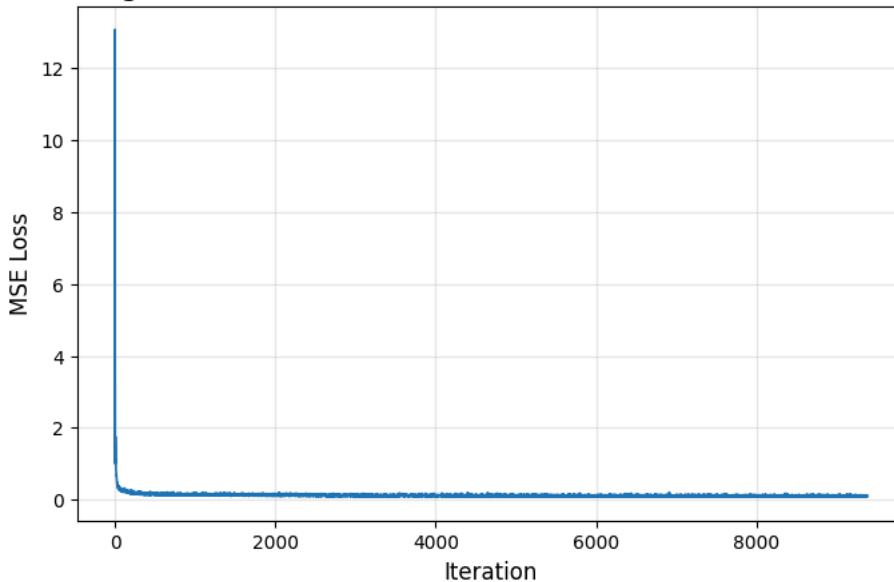
# ---- SAVE CHECKPOINTS for 1, 5, 10 ----
if epoch+1 in [1, 5, 10]:
    torch.save(unet.state_dict(), f"class_unet_epoch{epoch+1}.pth")
# ===== end of code =====

100%|██████████| 938/938 [00:42<00:00, 21.98it/s]
Epoch 1, loss = 0.1647
100%|██████████| 938/938 [00:42<00:00, 21.96it/s]
Epoch 2, loss = 0.1180
100%|██████████| 938/938 [00:42<00:00, 22.14it/s]
Epoch 3, loss = 0.1442
100%|██████████| 938/938 [00:42<00:00, 22.13it/s]
Epoch 4, loss = 0.0931
100%|██████████| 938/938 [00:42<00:00, 22.16it/s]
Epoch 5, loss = 0.1593
100%|██████████| 938/938 [00:42<00:00, 22.11it/s]
Epoch 6, loss = 0.1139
100%|██████████| 938/938 [00:42<00:00, 22.13it/s]
Epoch 7, loss = 0.0802
100%|██████████| 938/938 [00:42<00:00, 22.08it/s]
Epoch 8, loss = 0.1353
100%|██████████| 938/938 [00:42<00:00, 22.13it/s]
Epoch 9, loss = 0.1447
100%|██████████| 938/938 [00:42<00:00, 22.06it/s]Epoch 10, loss = 0.0971
```

```
import matplotlib.pyplot as plt
import numpy as np

plt.figure(figsize=(8, 5))
plt.plot(train_losses, linewidth=1.2)
plt.title("Training Loss Curve (Class-Conditional UNet without scheduler)", fontsize=16)
plt.xlabel("Iteration", fontsize=12)
plt.ylabel("MSE Loss", fontsize=12)
plt.grid(alpha=0.3)
plt.show()
```

Training Loss Curve (Class-Conditional UNet without scheduler)



Part 2.6: Sampling from the Class-conditioned UNet

```
# Sampling from the UNet
# ===== your code here! =====
def load_unet_for_epoch(epoch):
    unet = ClassConditionalUNet(
        in_channels=1,
        num_classes=10,
        num_hiddens=64
    ).to(device)

    unet.load_state_dict(torch.load(f"class_unet_epoch{epoch}.pth", map_location=device))
    unet.eval()
    return unet
# ===== end of code =====
```

```
def sample_digits(unet, num_samples_per_class=4):
    device = next(unet.parameters()).device

    # create labels 0-9 repeated num_samples_per_class times
    labels = torch.arange(10, device=device).repeat_interleave(num_samples_per_class)

    samples = class_fm_sample(
        unet,
        labels,
        img_wh=(28, 28),
        num_ts=50,
        guidance_scale=5.0,
        seed=0
    )
    return samples # shape: (10*num_per_class, 1, 28, 28)
```

```
def visualize_digit_grid_with_spacing(samples, num_per_class=4, spacing=0.2):
    """
    samples: (40, 1, 28, 28)
    expected ordering: 0000, 1111, ..., 9999
    output grid:
        row 1 = digits 0-9
        row 2 = digits 0-9
    ...
    """

    samples = samples.cpu()
    num_classes = 10
    total = num_classes * num_per_class
    assert samples.shape[0] == total

    # ---- REORDER INTO GRID SHAPE ----
    reordered = []
    for r in range(num_per_class): # rows
        for d in range(num_classes): # digits
            reordered.append(samples[d * num_per_class + r])
```

```
reordered = torch.stack(reordered)

# ---- PLOT ----
fig, axs = plt.subplots(num_per_class, num_classes,
                       figsize=(num_classes, num_per_class),
                       squeeze=False)

# small but visible spacing
plt.subplots_adjust(wspace=spacing, hspace=spacing)

idx = 0
for i in range(num_per_class):
    for j in range(num_classes):
        axs[i, j].imshow(reordered[idx, 0], cmap="gray")
        axs[i, j].axis("off")
        idx += 1

plt.show()
```

```
unet = load_unet_for_epoch(1)
samples, _ = sample_digits(unet, num_samples_per_class=4)
visualize_digit_grid_with_spacing(samples, num_per_class=4)
```

