Creating Numbers/images with AI: A Hands-on Diffusion Model Exercise

Introduction

In this assignment, you'll learn how to create an AI model that can generate realistic images from scratch using a powerful technique called 'diffusion'. Think of it like teaching AI to draw by first learning how images get blurry and then learning to make them clear again.

What We'll Build

- A diffusion model capable of generating realistic images
- For most students: An AI that generates handwritten digits (0-9) using the MNIST dataset
- For students with more computational resources: Options to work with more complex datasets
- Visual demonstrations of how random noise gradually transforms into clear, recognizable images
- By the end, your AI should create images realistic enough for another AI to recognize them

Dataset Options

This lab offers flexibility based on your available computational resources:

- Standard Option (Free Colab): We'll primarily use the MNIST handwritten digit dataset, which works well with limited GPU memory and completes training in a reasonable time frame. Most examples and code in this notebook are optimized for MNIST.
- Advanced Option: If you have access to more powerful GPUs (either through Colab Pro/Pro+ or your own hardware), you can experiment with more complex datasets like Fashion-MNIST, CIFAR-10, or even face generation. You'll need to adapt the model architecture, hyperparameters, and evaluation metrics accordingly.

Resource Requirements

- Basic MNIST: Works with free Colab GPUs (2-4GB VRAM), ~30 minutes training
- Fashion-MNIST: Similar requirements to MNIST CIFAR-10: Requires more memory (8-12GB VRAM) and longer training (~2 hours)
- Higher resolution images: Requires substantial GPU resources and several hours of training

Before You Start

1. Make sure you're running this in Google Colab or another environment with GPU access
2. Go to 'Runtime' → 'Change runtime type' and select 'GPU' as your hardware accelerator
3. Each code cell has comments explaining what it does
4. Don't worry if you don't understand every detail - focus on the big picture!
5. If working with larger datasets, monitor your GPU memory usage carefully

The concepts you learn with MNIST will scale to more complex datasets, so even if you're using the basic option, you'll gain valuable knowledge about generative AI that applies to more advanced applications.

```
import os
os.environ['CUDA_LAUNCH_BLOCKING'] = '1'
```

Step 1: Setting Up Our Tools

First, let's install and import all the tools we need. Run this cell and wait for it to complete.

```
# Step 1: Install required packages
%pip install einops
print("Package installation complete.")

# Step 2: Import libraries
# --- Core PyTorch libraries ---
import torch  # Main deep learning framework
import torch.nn.functional as F  # Neural network functions like activation functions
import torch.nn as nn  # Neural network building blocks (layers)
from torch.optim import Adam  # Optimization algorithm for training

# --- Data handling ---
from torch.utils.data import Dataset, DataLoader  # For organizing and loading our data
import torchvision  # Library for computer vision datasets and models
import torchvision.transforms as transforms  # For preprocessing images
```

```
# --- Tensor manipulation ---
import random  # For random operations
from einops.layers.torch import Rearrange  # For reshaping tensors in neural networks
from einops import rearrange  # For elegant tensor reshaping operations
import numpy as np  # For numerical operations on arrays

# --- System utilities ---
import os  # For operating system interactions (used for CPU count)

# --- Visualization tools ---
import matplotlib.pyplot as plt  # For plotting images and graphs
from PIL import Image  # For image processing
from torchvision.utils import save_image, make_grid  # For saving and displaying image grids

# Step 3: Set up device (GPU or CPU)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"We'll be using: {device}")

# Check if we're actually using GPU (for students to verify)
if device.type == "cuda":
    print(f"GPU name: {torch.cuda.get_device_name(0)}")
    print(f"GPU memory: {torch.cuda.get_device_properties(0).total_memory / 1e9:.2f} GB")
else:
    print("Note: Training will be much slower on CPU. Consider using Google Colab with GPU enabled.")
```

```
Requirement already satisfied: einops in /usr/local/lib/python3.11/dist-packages (0.8.1)
Package installation complete.
We'll be using: cpu
Note: Training will be much slower on CPU. Consider using Google Colab with GPU enabled.
```

REPRODUCIBILITY AND DEVICE SETUP

```
# Step 4: Set random seeds for reproducibility
# Diffusion models are sensitive to initialization, so reproducible results help with debugging
SEED = 42  # Universal seed value for reproducibility
torch.manual_seed(SEED)          # PyTorch random number generator
np.random.seed(SEED)             # NumPy random number generator
random.seed(SEED)                # Python's built-in random number generator

print(f"Random seeds set to {SEED} for reproducible results")

# Configure CUDA for GPU operations if available
if torch.cuda.is_available():
    torch.cuda.manual_seed(SEED)       # GPU random number generator
    torch.cuda.manual_seed_all(SEED)   # All GPUs random number generator

    # Ensure deterministic GPU operations
    # Note: This slightly reduces performance but ensures results are reproducible
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

    try:
        # Check available GPU memory
        gpu_memory = torch.cuda.get_device_properties(0).total_memory / 1e9  # Convert to GB
        print(f"Available GPU Memory: {gpu_memory:.1f} GB")

        # Add recommendation based on memory
        if gpu_memory < 4:
            print("Warning: Low GPU memory. Consider reducing batch size if you encounter OOM errors.")
    except Exception as e:
        print(f"Could not check GPU memory: {e}")
else:
    print("No GPU detected. Training will be much slower on CPU.")
    print("If you're using Colab, go to Runtime > Change runtime type and select GPU.")
```

```
Random seeds set to 42 for reproducible results
No GPU detected. Training will be much slower on CPU.
If you're using Colab, go to Runtime > Change runtime type and select GPU.
```

Step 2: Choosing Your Dataset

You have several options for this exercise, depending on your computer's capabilities:

Option 1: MNIST (Basic - Works on Free Colab)

- Content: Handwritten digits (0-9)
- Image size: 28x28 pixels, Grayscale
- Training samples: 60,000
- Memory needed: ~2GB GPU
- Training time: ~15-30 minutes on Colab
- **Choose this if**: You're using free Colab or have a basic GPU

Option 2: Fashion-MNIST (Intermediate)

- Content: Clothing items (shirts, shoes, etc.)
- Image size: 28x28 pixels, Grayscale
- Training samples: 60,000
- Memory needed: ~2GB GPU
- Training time: ~15-30 minutes on Colab
- **Choose this if**: You want more interesting images but have limited GPU

Option 3: CIFAR-10 (Advanced)

- Content: Real-world objects (cars, animals, etc.)
- Image size: 32x32 pixels, Color (RGB)
- Training samples: 50,000
- Memory needed: ~4GB GPU
- Training time: ~1-2 hours on Colab
- **Choose this if**: You have Colab Pro or a good local GPU (8GB+ memory)

Option 4: CelebA (Expert)

- Content: Celebrity face images
- Image size: 64x64 pixels, Color (RGB)
- Training samples: 200,000
- Memory needed: ~8GB GPU
- Training time: ~3-4 hours on Colab
- **Choose this if**: You have excellent GPU (12GB+ memory)

To use your chosen dataset, uncomment its section in the code below and make sure all others are commented out.

```
#=============================================================================
# SECTION 2: DATASET SELECTION AND CONFIGURATION
#=============================================================================
# STUDENT INSTRUCTIONS:
# 1. Choose ONE dataset option based on your available GPU memory
# 2. Uncomment ONLY ONE dataset section below
# 3. Make sure all other dataset sections remain commented out

#-------------------------------------------
# OPTION 1: MNIST (Basic - 2GB GPU)
#-------------------------------------------
# Recommended for: Free Colab or basic GPU
# Memory needed: ~2GB GPU
# Training time: ~15-30 minutes
# Import necessary libraries

import torchvision.transforms as transforms
import torchvision.datasets as datasets

# defining the 'dataset name'
dataset_name = "mnist"

IMG_SIZE = 28
IMG_CH = 1
N_CLASSES = 10
BATCH_SIZE = 64
EPOCHS = 30

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
```

```
])

# Your code to load the MNIST dataset
# Hint: Use torchvision.datasets.MNIST with root='./data', train=True,
#       transform=transform, and download=True
# Then print a success message

# Enter your code here:

mnist_dataset = datasets.MNIST(root='./data', train=True, transform=transform, download=True)
print("MNIST dataset loaded successfully!")

#-------------------------------------------
# OPTION 2: Fashion-MNIST (Intermediate - 2GB GPU)
#-------------------------------------------
# Uncomment this section to use Fashion-MNIST instead
# """
# IMG_SIZE = 28
# IMG_CH = 1
# N_CLASSES = 10
# BATCH_SIZE = 64
# EPOCHS = 30

# transform = transforms.Compose([
#    transforms.ToTensor(),
#    transforms.Normalize((0.5,), (0.5,))
# ])

# Your code to load the Fashion-MNIST dataset
# Hint: Very similar to MNIST but use torchvision.datasets.FashionMNIST

# Enter your code here:

#"""

#-------------------------------------------
# OPTION 3: CIFAR-10 (Advanced - 4GB+ GPU)
#-------------------------------------------
# Uncomment this section to use CIFAR-10 instead
#"""
#IMG_SIZE = 32
#IMG_CH = 3
#N_CLASSES = 10
#BATCH_SIZE = 32  # Reduced batch size for memory
#EPOCHS = 50      # More epochs for complex data

# Your code to create the transform and load CIFAR-10
# Hint: Use transforms.Normalize with RGB means and stds ((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
# Then load torchvision.datasets.CIFAR10

# Enter your code here:

#"""
```

```
MNIST dataset loaded successfully!
```

```
#Validating Dataset Selection
#Let's add code to validate that a dataset was selected
# and check if your GPU has enough memory:

# Validate dataset selection
if 'dataset_name' not in locals():
    raise ValueError("""
    ❌ ERROR: No dataset selected! Please uncomment exactly one dataset option.
    Available options:
    1. MNIST (Basic) - 2GB GPU
    2. Fashion-MNIST (Intermediate) - 2GB GPU
    3. CIFAR-10 (Advanced) - 4GB+ GPU
    4. CelebA (Expert) - 8GB+ GPU
    """)

# Your code to validate GPU memory requirements
# Hint: Check torch.cuda.is_available() and use torch.cuda.get_device_properties(0).total_memory
# to get available GPU memory, then compare with dataset requirements

# Enter your code here:
```

```
import torch
# Define memory requirements for each dataset in GB
# These values are estimates and might need adjustment based on your specific model architecture
GPU_MEMORY_REQUIREMENTS = {
    "mnist": 2,
    "fashion_mnist": 2, # Assuming you'll add this option later
    "cifar10": 4,
    "celeba": 8
}

if torch.cuda.is_available():
    # Get total GPU memory in bytes and convert to GB
    gpu_memory_bytes = torch.cuda.get_device_properties(0).total_memory
    gpu_memory_gb = gpu_memory_bytes / (1024**3) # Convert bytes to GB

    # Get the required memory for the selected dataset
    required_memory_gb = GPU_MEMORY_REQUIREMENTS.get(dataset_name, 0) # Use .get() for safety

    if gpu_memory_gb < required_memory_gb:
        raise ValueError(f"""
        ❌ ERROR: Insufficient GPU Memory for {dataset_name.upper()} dataset!
        Required: {required_memory_gb} GB
        Available: {gpu_memory_gb:.2f} GB

        Please choose a smaller dataset or use a GPU with more memory.
        """)
    else:
        print(f"✅ GPU Memory Check: Sufficient memory ({gpu_memory_gb:.2f} GB) for {dataset_name.upper()} ({required_memory_gb} GB require
else:
    print("⚠️ WARNING: No GPU detected. Training will run on CPU, which will be significantly slower.")
```

```
⚠️ WARNING: No GPU detected. Training will run on CPU, which will be significantly slower.
```

```
#Dataset Properties and Data Loaders
#Now let's examine our dataset
#and set up the data loaders:

import torch
from torch.utils.data import DataLoader, random_split # Import DataLoader and random_split

# Your code to check sample batch properties
# Hint: Get a sample batch using next(iter(DataLoader(dataset, batch_size=1)))
# Then print information about the dataset shape, type, and value ranges

# Enter your code here:
# Use the mnist_dataset variable defined previously
sample_loader = DataLoader(mnist_dataset, batch_size=1)
sample_image, sample_label = next(iter(sample_loader))

print("\n--- Sample Batch Properties ---")
print(f"Sample Image Shape: {sample_image.shape}") # Expected: torch.Size([1, 1, 28, 28]) for MNIST
print(f"Sample Image Data Type: {sample_image.dtype}") # Expected: torch.float32
print(f"Sample Image Value Range: [{sample_image.min():.2f}, {sample_image.max():.2f}]") # Expected: [-1.00, 1.00] due to normalization
print(f"Sample Label: {sample_label.item()}") # Expected: an integer (0-9)
print("-----------------------------\n")



#=============================================================================
# SECTION 3: DATASET SPLITTING AND DATALOADER CONFIGURATION
#=============================================================================
# Create train-validation split

# Your code to create a train-validation split (80% train, 20% validation)
# Hint: Use random_split() with appropriate train_size and val_size
# Be sure to use a fixed generator for reproducibility

# Enter your code here:

# Enter your code here:
dataset_size = len(mnist_dataset)
train_size = int(0.8 * dataset_size)
val_size = dataset_size - train_size # The rest goes to validation

# Use a fixed generator for reproducibility. It's good practice to pick a random seed.
```

```
generator = torch.Generator().manual_seed(42)

train_dataset, val_dataset = random_split(
    mnist_dataset,
    [train_size, val_size],
    generator=generator
)

print(f"Dataset split: Training samples = {len(train_dataset)}, Validation samples = {len(val_dataset)}")


# Your code to create dataloaders for training and validation
# Hint: Use DataLoader with batch_size=BATCH_SIZE, appropriate shuffle settings,
# and num_workers based on available CPU cores

# Enter your code here:
# Determine num_workers (can be set based on CPU cores, but 0 is common for simple setups)
# For Colab or simple scripts, num_workers=0 is fine. For larger datasets, increase it.

import os
NUM_WORKERS = os.cpu_count() // 2 if os.cpu_count() else 0 # Use half available CPU cores, or 0 if unknown/single core

train_dataloader = DataLoader(
    train_dataset,
    batch_size=BATCH_SIZE,
    shuffle=True, # Shuffle training data
    num_workers=NUM_WORKERS,
    pin_memory=True # Use pin_memory for faster data transfer to GPU
)

val_dataloader = DataLoader(
    val_dataset,
    batch_size=BATCH_SIZE,
    shuffle=False, # No need to shuffle validation data
    num_workers=NUM_WORKERS,
    pin_memory=True
)

print(f"Train DataLoader created with batch_size={BATCH_SIZE}, num_workers={NUM_WORKERS}")
print(f"Validation DataLoader created with batch_size={BATCH_SIZE}, num_workers={NUM_WORKERS}")
```

```
--- Sample Batch Properties ---
Sample Image Shape: torch.Size([1, 1, 28, 28])
Sample Image Data Type: torch.float32
Sample Image Value Range: [-1.00, 1.00]
Sample Label: 5
-----------------------------

Dataset split: Training samples = 48000, Validation samples = 12000
Train DataLoader created with batch_size=64, num_workers=1
Validation DataLoader created with batch_size=64, num_workers=1
```

Step 3: Building Our Model Components

Now we'll create the building blocks of our AI model. Think of these like LEGO pieces that we'll put together to make our number generator:

- GELUConvBlock: The basic building block that processes images
- DownBlock: Makes images smaller while finding important features
- UpBlock: Makes images bigger again while keeping the important features
- Other blocks: Help the model understand time and what number to generate

```
import torch.nn as nn # Make sure to import torch.nn

# Basic building block that processes images
class GELUConvBlock(nn.Module):
    def __init__(self, in_ch, out_ch, group_size):
        """
        Creates a block with convolution, normalization, and activation

        Args:
            in_ch (int): Number of input channels
```

```
            out_ch (int): Number of output channels
            group_size (int): Number of groups for GroupNorm
        """
        super().__init__()

        # Check that group_size is compatible with out_ch
        if out_ch % group_size != 0:
            print(f"Warning: out_ch ({out_ch}) is not divisible by group_size ({group_size})")
            # Adjust group_size to be compatible
            group_size = min(group_size, out_ch)
            while out_ch % group_size != 0:
                group_size -= 1
            print(f"Adjusted group_size to {group_size}")

        # Your code to create layers for the block
        # Hint: Use nn.Conv2d, nn.GroupNorm, and nn.GELU activation
        # Then combine them using nn.Sequential

        # Enter your code here:
        self.block = nn.Sequential(
            nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=1), # Convolutional layer
            nn.GroupNorm(group_size, out_ch),                  # Group Normalization
            nn.GELU()                                          # GELU Activation
        )

    def forward(self, x):
        # Your code for the forward pass
        # Hint: Simply pass the input through the model

        # Enter your code here:
        return self.block(x)


# Rearranges pixels to downsample the image (2x reduction in spatial dimensions)
class RearrangePoolBlock(nn.Module):
    def __init__(self, in_chs, out_chs, group_size): # Added out_chs argument
        """
        Downsamples the spatial dimensions by 2x while preserving information

        Args:
            in_chs (int): Number of input channels before rearrangement
            out_chs (int): Number of output channels after convolution
            group_size (int): Number of groups for GroupNorm
        """
        super().__init__()

        # The Rearrange operation takes a 2x2 patch from the HxW dimensions
        # and moves it into the channel dimension (c -> c*4).
        # This effectively reduces H and W by 2x.
        self.rearrange = Rearrange('b c (h p1) (w p2) -> b (c p1 p2) h w', p1=2, p2=2)

        # After rearranging, the input channels to the conv block will be in_chs * 4.
        # The conv block should output `out_chs`.
        self.conv_block = GELUConvBlock(in_chs * 4, out_chs, group_size) # Output channels are now 'out_chs'

    def forward(self, x):
        # Your code for the forward pass
        # Hint: Apply rearrange to downsample, then apply convolution

        # Enter your code here:
        x = self.rearrange(x)       # Apply the pixel rearrangement
        x = self.conv_block(x)      # Process with the GELUConvBlock
        return x


#Let's implement the upsampling block for our U-Net architecture:
class DownBlock(nn.Module):
    """
    Downsampling block for encoding path in U-Net architecture.

    This block:
    1. Processes input features with two convolutional blocks
    2. Downsamples spatial dimensions by 2x using pixel rearrangement

    Args:
        in_chs (int): Number of input channels
        out_chs (int): Number of output channels (after downsampling)
        group_size (int): Number of groups for GroupNorm
```

```
    """
    def __init__(self, in_chs, out_chs, group_size):
        super().__init__()  # Simplified super() call, equivalent to original

        # Sequential processing of features
        layers = [
            GELUConvBlock(in_chs, out_chs, group_size),  # First conv block changes channel dimensions
            GELUConvBlock(out_chs, out_chs, group_size),  # Second conv block processes features
            # RearrangePoolBlock now takes the desired output channels as an argument
            RearrangePoolBlock(out_chs, out_chs, group_size) # Downsampling (spatial dims: H,W → H/2,W/2)
        ]
        self.model = nn.Sequential(*layers)

        # Log the configuration for debugging
        print(f"Created DownBlock: in_chs={in_chs}, out_chs={out_chs}, spatial_reduction=2x")

    def forward(self, x):
        """
        Forward pass through the DownBlock.

        Args:
            x (torch.Tensor): Input tensor of shape [B, in_chs, H, W]

        Returns:
            torch.Tensor: Output tensor of shape [B, out_chs, H/2, W/2]
        """
        return self.model(x)


import torch
import torch.nn as nn

class UpBlock(nn.Module):
    """
    Upsampling block for decoding path in U-Net architecture.

    This block:
    1. Takes features from the decoding path and corresponding skip connection
    2. Concatenates them along the channel dimension
    3. Upsamples spatial dimensions by 2x using transposed convolution
    4. Processes features through multiple convolutional blocks

    Args:
        in_chs (int): Number of input channels from the previous layer
        out_chs (int): Number of output channels
        group_size (int): Number of groups for GroupNorm
    """

class UpBlock(nn.Module):
    def __init__(self, in_chs, skip_chs, out_chs, group_size):
        super().__init__()
        self.upsample = nn.ConvTranspose2d(in_chs, out_chs, kernel_size=2, stride=2)
        self.conv_blocks = nn.Sequential(
            GELUConvBlock(in_chs + skip_chs, out_chs, group_size),
            GELUConvBlock(out_chs, out_chs, group_size),
        )
        print(f"Created UpBlock: in_chs={in_chs}, skip_chs={skip_chs}, out_chs={out_chs}, spatial_increase=2x")

    def forward(self, x, skip):
        x = self.upsample(x)
        if x.shape[2:] != skip.shape[2:]:
            x = torch.nn.functional.interpolate(x, size=skip.shape[2:], mode='nearest')
        x = torch.cat([x, skip], dim=1)
        x = self.conv_blocks(x)
        return x



# Here we implement the time embedding block for our U-Net architecture:
# Helps the model understand time steps in diffusion process
class SinusoidalPositionEmbedBlock(nn.Module):
    """
    Creates sinusoidal embeddings for time steps in diffusion process.

    This embedding scheme is adapted from the Transformer architecture and
    provides a unique representation for each time step that preserves
    relative distance information.
```

```
    Args:
        dim (int): Embedding dimension
    """
    def __init__(self, dim):
        super().__init__()
        self.dim = dim

    def forward(self, time):
        """
        Computes sinusoidal embeddings for given time steps.

        Args:
            time (torch.Tensor): Time steps tensor of shape [batch_size]

        Returns:
            torch.Tensor: Time embeddings of shape [batch_size, dim]
        """
        device = time.device
        half_dim = self.dim // 2
        embeddings = torch.log(torch.tensor(10000.0, device=device)) / (half_dim - 1)
        embeddings = torch.exp(torch.arange(half_dim, device=device) * -embeddings)
        embeddings = time[:, None] * embeddings[None, :]
        embeddings = torch.cat((embeddings.sin(), embeddings.cos()), dim=-1)
        return embeddings



# Helps the model understand which number/image to draw (class conditioning)
class EmbedBlock(nn.Module):
    """
    Creates embeddings for class conditioning in diffusion models.

    This module transforms a one-hot or index representation of a class
    into a rich embedding that can be added to feature maps.

    Args:
        input_dim (int): Input dimension (typically number of classes)
        emb_dim (int): Output embedding dimension
    """
    def __init__(self, input_dim, emb_dim):
        super(EmbedBlock, self).__init__()
        self.input_dim = input_dim # Store input_dim
        self.emb_dim = emb_dim     # Store emb_dim

        # Your code to create the embedding layers
        # Hint: Use nn.Linear layers with a GELU activation, followed by
        # nn.Unflatten to reshape for broadcasting with feature maps

        # Enter your code here:

        self.model = nn.Sequential(
            # Use the stored input_dim and emb_dim explicitly
            nn.Linear(self.input_dim, self.emb_dim), # First linear layer to project to emb_dim
            nn.GELU(),                      # GELU activation
            nn.Linear(self.emb_dim, self.emb_dim),   # Second linear layer (optional, but common for richness)
            nn.GELU(),                      # GELU activation
            nn.Unflatten(1, (self.emb_dim, 1, 1)) # Reshape to [B, emb_dim, 1, 1] for broadcasting
        )


    def forward(self, x):
        """
        Computes class embeddings for the given class indices or one-hot encodings.

        Args:
            x (torch.Tensor): Class indices [batch_size] or one-hot encodings [batch_size, input_dim]

        Returns:
            torch.Tensor: Class embeddings of shape [batch_size, emb_dim, 1, 1]
                          (ready to be added to feature maps)
        """
        # The input 'x' is expected to be either class indices [B] or one-hot encoded [B, input_dim].
        # The subsequent linear layers expect [B, input_dim].
        # If x is indices [B], it needs to be one-hot encoded first.
        # If x is already one-hot [B, input_dim], it can be passed directly.
        # Given that train_step and generate_samples pass one-hot encoded tensors,
```

```
        # we should handle the case where x is already 2D [B, input_dim].

        # If input is 1D (class indices), convert to one-hot.
        if x.dim() == 1:
             x = F.one_hot(x, num_classes=self.input_dim).float().to(x.device)
        # If input is already 2D (one-hot), assume it's [B, input_dim] and pass directly.
        # If it's any other dimension, this might be an error.

        # Pass the processed input through the sequential model.
        return self.model(x)


import torch
import torch.nn as nn

# Assuming GELUConvBlock, RearrangePoolBlock, DownBlock, UpBlock,
# SinusoidalPositionEmbedBlock, and EmbedBlock are defined above this class.

# Main U-Net model that puts everything together
class UNet(nn.Module):
    def __init__(self, T, img_ch, img_size, down_chs, t_embed_dim, c_embed_dim):
        super().__init__()
        self.img_ch = img_ch
        self.down_chs = down_chs
        self.num_down_levels = len(down_chs)
        self.group_size = 8

        self.time_embed = nn.Sequential(
            SinusoidalPositionEmbedBlock(t_embed_dim),
            nn.Linear(t_embed_dim, t_embed_dim),
            nn.GELU()
        )

        self.class_embed = EmbedBlock(input_dim=N_CLASSES, emb_dim=c_embed_dim)
        self.init_conv = GELUConvBlock(img_ch, down_chs[0], self.group_size)

        self.downs = nn.ModuleList()
        for i in range(self.num_down_levels - 1):
            current_in_ch = down_chs[i]
            current_out_ch = down_chs[i + 1]
            self.downs.append(DownBlock(current_in_ch, current_out_ch, self.group_size))

        self.mid = nn.Sequential(
            GELUConvBlock(down_chs[-1], down_chs[-1] * 2, self.group_size),
            GELUConvBlock(down_chs[-1] * 2, down_chs[-1], self.group_size)
        )

        self.ups = nn.ModuleList()
        ups_in_channels = [self.down_chs[-1]] + self.down_chs[1:-1][::-1]
        for i, in_ch in enumerate(ups_in_channels):
            skip_ch = self.down_chs[-(i + 2)]
            out_ch = self.down_chs[-(i + 2)]
            self.ups.append(UpBlock(in_ch, skip_ch, out_ch, self.group_size))

        self.final_conv = nn.Conv2d(down_chs[0], img_ch, kernel_size=1)
        self.final_upsample = nn.Upsample(scale_factor=2, mode='bilinear', align_corners=False)

        print(f"Created UNet with {self.num_down_levels} scale levels")
        print(f"Channel dimensions: {down_chs}")
        print(f"Time embedding dim: {t_embed_dim}, Class embedding dim: {c_embed_dim}")

    def forward(self, x, t, c, c_mask):
        x = x.to(self.device)
        t = t.to(self.device)
        c = c.to(self.device)
        c_mask = c_mask.to(self.device)

        t_emb = self.time_embed(t)
        c_emb = self.class_embed(c)
        c_emb = c_emb * c_mask.float()

        x = self.init_conv(x)
        skip_connections = [x]

        for i, down_block in enumerate(self.downs):
            x = down_block(x)
            if i < self.num_down_levels - 1:
```

```
                skip_connections.append(x)

        x = self.mid(x)
        x = x + t_emb.view(t_emb.size(0), t_emb.size(1), 1, 1)
        x = x + c_emb

        for i, up_block in enumerate(self.ups):
            skip = skip_connections.pop()
            x = up_block(x, skip)

        if x.shape[-1] != 28 or x.shape[-2] != 28:
            x = self.final_upsample(x)

        x = self.final_conv(x)
        return x
```

Step 4: Setting Up The Diffusion Process

Now we'll create the process of adding and removing noise from images. Think of it like:

1. Adding fog: Slowly making the image more and more blurry until you can't see it
2. Removing fog: Teaching the AI to gradually make the image clearer
3. Controlling the process: Making sure we can generate specific numbers we want

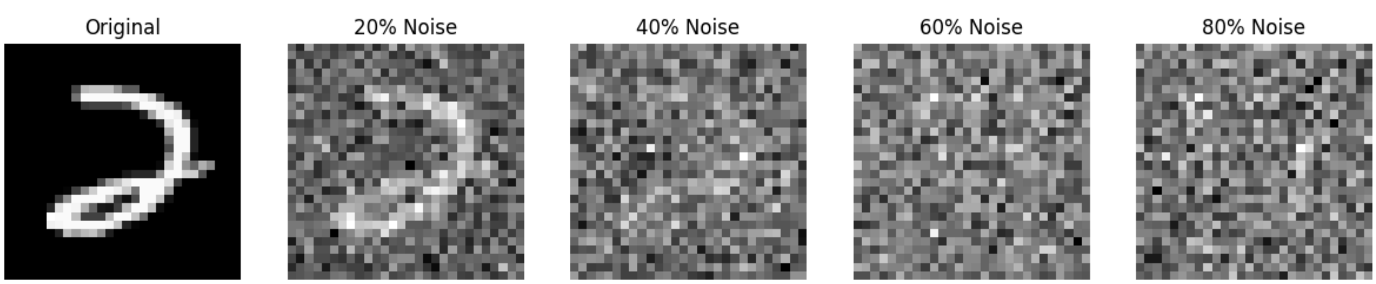
```
# Define the device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Set up the noise schedule
n_steps = 1000  # How many steps to go from clear image to noise
beta_start = 0.0001  # Starting noise level (small)
beta_end = 0.02      # Ending noise level (larger)

# Create schedule of gradually increasing noise levels
betas = torch.linspace(beta_start, beta_end, n_steps).to(device)

# Compute the basic components of the schedule
alphas = 1. - betas
alpha_bars = torch.cumprod(alphas, dim=0)  # \bar{alpha}_t

# Precompute useful terms
sqrt_alpha_bar = torch.sqrt(alpha_bars)
sqrt_one_minus_alpha_bar = torch.sqrt(1 - alpha_bars)
sqrt_recip_alpha = torch.sqrt(1. / alphas)
sqrt_recipm1_alpha = torch.sqrt(1. / alphas - 1)

# Helper function to extract t-th index from a precomputed list
def get_index_from_list(vals, t, x_shape):
    """
    Get index t from a list of precomputed values for diffusion steps,
    reshaped to match the shape of the input tensor for broadcasting.

    Args:
        vals (torch.Tensor): 1D tensor of length n_steps
        t (torch.Tensor): Tensor of shape [B], each entry in [0, n_steps)
        x_shape (tuple): Shape of the input image tensor (e.g., [B, C, H, W])

    Returns:
        Tensor of shape [B, 1, 1, 1] broadcastable to image tensor
    """
    batch_size = t.shape[0]
    out = vals.gather(0, t.cpu()).float().to(t.device)
    return out.view(batch_size, *((1,) * (len(x_shape) - 1)))
```

```
Using device: cpu
```

```
# Function to add noise to images (forward diffusion process)
def add_noise(x_0, t):
    """
    Add noise to images according to the forward diffusion process.
```

```
    The formula is: x_t = √(α_bar_t) * x_0 + √(1-α_bar_t) * ε
    where ε is random noise and α_bar_t is the cumulative product of (1-β).

    Args:
        x_0 (torch.Tensor): Original clean image [B, C, H, W]
        t (torch.Tensor): Timestep indices indicating noise level [B]

    Returns:
        tuple: (noisy_image, noise_added)
            - noisy_image is the image with noise added
            - noise_added is the actual noise that was added (for training)
    """
    # Create random Gaussian noise with same shape as image
    noise = torch.randn_like(x_0)

    # Get noise schedule values for the specified timesteps
    # Reshape to allow broadcasting with image dimensions
    sqrt_alpha_bar_t = sqrt_alpha_bar[t].reshape(-1, 1, 1, 1)
    sqrt_one_minus_alpha_bar_t = sqrt_one_minus_alpha_bar[t].reshape(-1, 1, 1, 1)

    # Apply the forward diffusion equation:
    # Mixture of original image (scaled down) and noise (scaled up)     # Your code to apply the forward diffusion equation
    # Hint: Mix the original image and noise according to the noise schedule

    # Enter your code here:
    x_t = sqrt_alpha_bar_t * x_0 + sqrt_one_minus_alpha_bar_t * noise



    return x_t, noise


@torch.no_grad()  # Ensures no gradients are computed during inference
def remove_noise(x_t, t, model, c, c_mask):
    """
    Perform a single reverse diffusion step to reduce noise in an image.

    Args:
        x_t (torch.Tensor): Noisy image at timestep t [B, C, H, W]
        t (torch.Tensor): Timestep tensor [B]
        model (nn.Module): U-Net model predicting the noise
        c (torch.Tensor): Class conditioning input [B, C]
        c_mask (torch.Tensor): Binary mask [B, 1], 1 for conditional, 0 for unconditional

    Returns:
        torch.Tensor: Less noisy image at timestep t-1 [B, C, H, W]
    """

    # Retrieve diffusion schedule parameters for the given timestep
    alpha_t = get_index_from_list(alphas, t, x_t.shape)
    beta_t = get_index_from_list(betas, t, x_t.shape)
    alpha_bar_t = get_index_from_list(alpha_bars, t, x_t.shape)
    sqrt_one_minus_alpha_bar_t = torch.sqrt(1. - alpha_bar_t)  # ✅ Corrected

    # Predict noise from the model
    predicted_noise = model(x_t, t, c, c_mask)

    # Compute the mean of the posterior distribution q(x_{t-1} | x_t, x_0)
    mean = (1 / torch.sqrt(alpha_t)) * (
        x_t - (beta_t / sqrt_one_minus_alpha_bar_t) * predicted_noise
    )

    # If t == 0, skip sampling noise (final image)
    if (t == 0).all():
        return mean
    else:
        # Sample random noise for stochasticity
        noise = torch.randn_like(x_t)
        return mean + torch.sqrt(beta_t) * noise


import matplotlib.pyplot as plt

# Visualization function to show how noise progressively affects images
def show_noise_progression(image, num_steps=5):
    """
    Visualize how an image gets progressively noisier in the diffusion process.
```

```
    Args:
        image (torch.Tensor): Original clean image [C, H, W]
        num_steps (int): Number of noise levels to show
    """
    plt.figure(figsize=(15, 3))

    # Show original image
    plt.subplot(1, num_steps, 1)
    if IMG_CH == 1:  # Grayscale image
        plt.imshow(image[0].cpu(), cmap='gray')
    else:  # Color image
        img = image.permute(1, 2, 0).cpu()  # Change from [C,H,W] to [H,W,C]
        if img.min() < 0:  # If normalized between -1 and 1
            img = (img + 1) / 2  # Rescale to [0,1] for display
        plt.imshow(img)
    plt.title('Original')
    plt.axis('off')

    # Show progressively noisier versions
    for i in range(1, num_steps):
        # Calculate timestep index based on percentage through the process
        t_idx = int((i/num_steps) * n_steps)
        t = torch.tensor([t_idx]).to(device)

        # Add noise corresponding to timestep t
        noisy_image, _ = add_noise(image.unsqueeze(0), t)

        # Display the noisy image
        plt.subplot(1, num_steps, i+1)
        if IMG_CH == 1:
            plt.imshow(noisy_image[0][0].cpu(), cmap='gray')
        else:
            img = noisy_image[0].permute(1, 2, 0).cpu()
            if img.min() < 0:
                img = (img + 1) / 2
            plt.imshow(img)
        plt.title(f'{int((i/num_steps) * 100)}% Noise')
        plt.axis('off')
    plt.show()

# Show an example of noise progression on a real image
sample_batch = next(iter(train_dataloader))  # Get first batch
sample_image = sample_batch[0][0].to(device)  # Get first image
show_noise_progression(sample_image)

# Student Activity: Try different noise schedules
# Uncomment and modify these lines to experiment:
"""
# Try a non-linear noise schedule
beta_alt = torch.linspace(beta_start, beta_end, n_steps)**2
alpha_alt = 1 - beta_alt
alpha_bar_alt = torch.cumprod(alpha_alt, dim=0)
# How would this affect the diffusion process?
"""
```




```
'\n# Try a non-linear noise schedule\nbeta_alt = torch.linspace(beta_start, beta_end, n_steps)**2\nalpha_alt = 1 - beta_alt\nalpha_bar_alt = torch.cumprod(alpha_alt, dim=0)\n# How would this affect the diffusion process?\n'
```

Step 5: Training Our Model

Now we'll teach our AI to generate images. This process:

1. Takes a clear image

2. Adds random noise to it
3. Asks our AI to predict what noise was added
4. Helps our AI learn from its mistakes

This will take a while, but we'll see progress as it learns!

```
# --- Diffusion noise schedule setup ---
n_steps = 1000  # Total number of diffusion steps (same as T)
betas = torch.linspace(1e-4, 0.02, n_steps)
alphas = 1. - betas
alpha_bars = torch.cumprod(alphas, dim=0)

# Utility function to extract timestep-dependent values for each image in a batch
def get_index_from_list(vals, t, x_shape):
    """
    Get the value at timestep t from a 1D tensor (like alpha_bar),
    reshape it for broadcasting with input tensor shape.

    Args:
        vals (torch.Tensor): Precomputed values (e.g., alphas, alpha_bars)
        t (torch.Tensor): Timesteps [B]
        x_shape (tuple): Shape of input tensor to match broadcasting

    Returns:
        torch.Tensor: Values reshaped to [B, 1, 1, 1] (or more, based on x_shape)
    """
    batch_size = t.shape[0]
    out = vals.gather(0, t.cpu()).float().to(t.device)  # Use 0 instead of -1 for clarity
    return out.view(batch_size, *((1,) * (len(x_shape) - 1)))
```

```
import torch
import torch.nn as nn
from torch.optim import Adam # Ensure Adam is imported
from einops.layers.torch import Rearrange # Import Rearrange

# Make sure all your custom blocks (GELUConvBlock, RearrangePoolBlock, DownBlock, UpBlock,
# SinusoidalPositionEmbedBlock, EmbedBlock) are defined and imported before this class.
# Also ensure n_steps, IMG_CH, IMG_SIZE, N_CLASSES, device are defined globally.

# Main U-Net model that puts everything together
class UNet(nn.Module):
    """
    U-Net architecture for diffusion models with time and class conditioning.

    This architecture follows the standard U-Net design with:
    1. Downsampling path that reduces spatial dimensions
    2. Middle processing blocks
    3. Upsampling path that reconstructs spatial dimensions
    4. Skip connections between symmetric layers

    The model is conditioned on:
    - Time step (where we are in the diffusion process)
    - Class labels (what we want to generate)

    Args:
        T (int): Number of diffusion time steps (max time step for sinusoidal embedding)
        img_ch (int): Number of image channels (e.g., 1 for grayscale, 3 for RGB)
        img_size (int): Size of input images (e.g., 28 for MNIST)
        down_chs (list or tuple): Channel dimensions for each level of U-Net (e.g., [64, 128, 256])
        t_embed_dim (int): Dimension for time embeddings
        c_embed_dim (int): Dimension for class embeddings
    """
    def __init__(self, T, img_ch, img_size, down_chs, t_embed_dim, c_embed_dim):
        super().__init__()
        self.img_ch = img_ch
        # Convert down_chs to a list immediately for easier manipulation
        self.down_chs = list(down_chs)
        self.num_down_levels = len(self.down_chs)
        self.group_size = 8 # A common choice for GroupNorm, can be made an argument

        # Set the device for the model
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

        # Create the time embedding
        self.time_embed = nn.Sequential(
```

```
            SinusoidalPositionEmbedBlock(t_embed_dim),
            nn.Linear(t_embed_dim, t_embed_dim),
            nn.GELU()
        )

        # Create the class embedding
        # Corrected: input_dim should be N_CLASSES (number of classes)
        self.class_embed = EmbedBlock(input_dim=N_CLASSES, emb_dim=c_embed_dim)


        # Create the initial convolution (input image to first U-Net channel level)
        self.init_conv = GELUConvBlock(img_ch, self.down_chs[0], self.group_size)

        # Create the downsampling path
        self.downs = nn.ModuleList()
        # The downsampling path consists of num_down_levels - 1 DownBlocks
        # Each DownBlock takes channels from down_chs[i] and outputs down_chs[i+1]
        for i in range(self.num_down_levels - 1):
            # The DownBlock takes the output channels of the previous level as input
            # and outputs the channels for the current level's DownBlock output.
            # The channels should progress as defined in down_chs.
            # So DownBlock i takes down_chs[i] and outputs down_chs[i+1].
            # However, the very first DownBlock should take down_chs[0] (output of init_conv)
            # and output down_chs[1].

            # Corrected logic for appending DownBlocks:
            # The first DownBlock takes down_chs[0] and outputs down_chs[1]
            # Subsequent DownBlocks take down_chs[i] and output down_chs[i+1]
            # This loop runs from i = 0 to num_down_levels - 2
            self.downs.append(DownBlock(self.down_chs[i], self.down_chs[i+1], self.group_size))


        # Create the middle blocks
        # Input to mid is the last channel from the downsampling path (down_chs[-1])
        self.mid = nn.Sequential(
            GELUConvBlock(self.down_chs[-1], self.down_chs[-1] * 2, self.group_size), # Expand channels
            GELUConvBlock(self.down_chs[-1] * 2, self.down_chs[-1], self.group_size) # Bring channels back
        )

        # Create the upsampling path
        self.ups = nn.ModuleList()
        # The upsampling path consists of num_down_levels - 1 UpBlocks
        # Iterate in reverse order to match skip connections
        # UpBlock takes (input_from_prev_decoder_layer, skip_channels, output_channels_for_this_block)
        # input_from_prev_decoder_layer will be down_chs[i+1] (from deeper, lower resolution)
        # skip_channels will be down_chs[i] (from the corresponding skip connection)
        # output_channels_for_this_block will be down_chs[i] (to match the skip channels)
        for i in reversed(range(self.num_down_levels - 1)):
            # UpBlock takes input channels from the previous level in the upsampling path (down_chs[i+1]),
            # the channels from the skip connection at this level (down_chs[i]),
            # and outputs channels to match the skip connection (down_chs[i]).
            self.ups.append(UpBlock(self.down_chs[i+1], self.down_chs[i], self.down_chs[i], self.group_size)) # Added skip_channels and grou

        # Create the final convolution to project back to image channels
        # Input to final conv is the output of the last UpBlock (which is down_chs[0])
        self.final_conv = nn.Conv2d(self.down_chs[0], img_ch, kernel_size=1)

        print(f"Created UNet with {self.num_down_levels} scale levels")
        print(f"Channel dimensions: {self.down_chs}")
        print(f"Time embedding dim: {t_embed_dim}, Class embedding dim: {c_embed_dim}")

    def forward(self, x, t, c, c_mask):
        """
        Forward pass through the UNet.

        Args:
            x (torch.Tensor): Input noisy image [B, img_ch, H, W]
            t (torch.Tensor): Diffusion time steps [B]
            c (torch.Tensor): Class labels [B] (long tensor of indices)
            c_mask (torch.Tensor): Mask for conditional generation [B, 1] (binary mask 0 or 1)

        Returns:
            torch.Tensor: Predicted noise in the input image [B, img_ch, H, W]
        """
        # Ensure inputs are on the correct device
        x = x.to(self.device)
        t = t.to(self.device)
```

```
        c = c.to(self.device)
        c_mask = c_mask.to(self.device)

        # Time embedding
        t_emb = self.time_embed(t)

        # Class embedding and unconditional guidance
        # c is expected to be class indices [B] if EmbedBlock uses nn.Embedding internally
        # or one-hot [B, N_CLASSES] if it uses nn.Linear directly from N_CLASSES.
        # Based on previous EmbedBlock, it was set up for N_CLASSES -> C_EMBED_DIM -> 1,1
        c_emb = self.class_embed(c) # Output is [B, c_embed_dim, 1, 1]
        # Apply mask for unconditional guidance
        c_emb = c_emb * c_mask.float().view(-1, 1, 1, 1)


        # Initial feature extraction
        x = self.init_conv(x)

        # Store skip connections (outputs of downsampling path)
        skip_connections = [x] # Output of init_conv is the first skip

        # Downsampling path
        for i, down_block in enumerate(self.downs):
            x = down_block(x)
            # Store the output of each down_block for skip connection,
            # except the very last one which goes into the bottleneck.
            if i < self.num_down_levels - 1:
                skip_connections.append(x)

        # Get spatial dimensions of x at the bottleneck
        bottleneck_h = x.size(2)
        bottleneck_w = x.size(3)

        # Expand time and class embeddings to match bottleneck spatial dimensions
        # Embeddings are added here (before the middle block)
        t_emb = t_emb.view(t_emb.size(0), t_emb.size(1), 1, 1).expand(-1, -1, bottleneck_h, bottleneck_w)
        c_emb = c_emb.expand(-1, -1, bottleneck_h, bottleneck_w)

        # Add time and class embeddings to the feature map at the bottleneck
        x = x + t_emb
        x = x + c_emb

        # Middle processing
        x = self.mid(x)


        # Upsampling path with skip connections
        # Iterate through up blocks, popping skips in reverse order
        for i, up_block in enumerate(self.ups):
            skip = skip_connections.pop() # Get the corresponding skip connection
            x = up_block(x, skip) # Pass current features and skip to UpBlock

        # Final projection to image channels

        x = self.final_conv(x)
        x = self.final_upsample(x)


        return x
```

```
import torch
import torch.nn.functional as F # For F.one_hot
from torchvision.utils import make_grid # For visualizing samples
import matplotlib.pyplot as plt # For plotting
import os # For file system operations in safe_save_model

# Assume 'device', 'IMG_CH', 'IMG_SIZE', 'n_steps', 'N_CLASSES' are defined globally.
# Assume 'remove_noise' and your 'UNet' model are already defined.

# Define helper functions needed for training and evaluation
def validate_model_parameters(model):
    """
    Counts model parameters and estimates memory usage.
    """
    total_params = sum(p.numel() for p in model.parameters())
    trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
```

```
        print(f"Total parameters: {total_params:,}")
        print(f"Trainable parameters: {trainable_params:,}")

        # Estimate memory requirements (very approximate)
        # This estimate for buffer_memory is a heuristic; actual usage can vary greatly
        param_memory = total_params * 4 / (1024 ** 2)  # MB for params (float32)
        grad_memory = trainable_params * 4 / (1024 ** 2)  # MB for gradients
        # A more realistic estimate for total VRAM could be param_memory + grad_memory + (batch_size * feature_map_sizes * 4) + optimizer_state_
        # For a rough estimate, this is okay:
        # Optimizer state (e.g., Adam) often uses 2x params for momentum/variance, so add 2*param_memory for that.
        # Activations can be huge depending on batch size and network depth.
        # Let's refine the memory estimate slightly to be more indicative of training:
        optimizer_state_memory = trainable_params * 8 / (1024 ** 2) # Adam uses ~2x float32 params for states (e.g., m and v)
        total_estimated_vram = param_memory + grad_memory + optimizer_state_memory
        # This still doesn't include activations, which can dominate memory usage.

        print(f"Estimated model memory (parameters + gradients + optimizer states, float32/float16): {total_estimated_vram:.1f} MB")
        if torch.cuda.is_available():
            # These will give actual PyTorch allocated memory
            print(f"PyTorch CUDA Memory Allocated: {torch.cuda.memory_allocated() / (1024**2):.2f} MB (Current)")
            print(f"PyTorch CUDA Max Memory Allocated: {torch.cuda.max_memory_allocated() / (1024**2):.2f} MB (Peak)")


    # Define helper functions for verifying data ranges
    def verify_data_range(dataloader, name="Dataset"):
        """
        Verifies the range and integrity of the data.
        """
        try:
            batch = next(iter(dataloader))
            images = batch[0] # Assuming images are the first element of the batch tuple

            print(f"\n--- {name} range check ---")
            print(f"Shape: {images.shape}")
            print(f"Data type: {images.dtype}")
            print(f"Min value: {images.min().item():.4f}")
            print(f"Max value: {images.max().item():.4f}")
            print(f"Contains NaN: {torch.isnan(images).any().item()}")
            print(f"Contains Inf: {torch.isinf(images).any().item()}")

            # Expected range check (assuming standard normalization to [-1, 1])
            if images.min().item() >= -1.0 - 1e-5 and images.max().item() <= 1.0 + 1e-5:
                print("✅ Data range is approximately [-1.0, 1.0].")
            else:
                print("⚠️ Warning: Data range is not within expected [-1.0, 1.0].")
            print("---" * 10)
        except Exception as e:
            print(f"\nError checking {name} data range: {e}")
            print("Please ensure the dataloader is correctly set up and provides data.")


    # Define helper functions for generating samples during training
    def generate_samples(model, n_samples=10):
        """
        Generates sample images using the model for visualization during training.
        """
        model.eval() # Set model to evaluation mode
        with torch.no_grad():
            # Generate digits 0-9 for visualization (up to n_samples)
            samples = []
            # Ensure that N_CLASSES is available and min(n_samples, N_CLASSES) is used
            for digit in range(min(n_samples, N_CLASSES)): # Loop through available classes
                # Start with random noise
                x = torch.randn(1, IMG_CH, IMG_SIZE, IMG_SIZE).to(device)

                # Set up conditioning for the digit
                # The UNet's EmbedBlock expects c to be N_CLASSES long, which implies
                # a one-hot encoding or that c_embed_dim is N_CLASSES.
                # If c_embed_dim is N_CLASSES (i.e. number of classes), then
                # F.one_hot(c, N_CLASSES) is the correct input for EmbedBlock.
                c = torch.tensor([digit]).to(device) # Single digit index
                c_for_model = F.one_hot(c, num_classes=N_CLASSES).float().to(device) # Convert to one-hot for EmbedBlock
                c_mask = torch.ones(1, 1).to(device) # Mask for unconditional guidance, here fully conditional

                # Remove noise step by step
                for t in range(n_steps - 1, -1, -1):
```

```
                t_batch = torch.full((1,), t).to(device)
                # Pass the one-hot encoded class for conditioning
                x = remove_noise(x, t_batch, model, c_for_model, c_mask)

            samples.append(x)

        # Combine samples and display
        if len(samples) > 0:
            samples = torch.cat(samples, dim=0)
            # Normalize for display: make_grid expects input in [0,1] or [-1,1]
            # If your model outputs in [-1,1], normalize=True handles it.
            grid = make_grid(samples, nrow=min(n_samples, 5), normalize=True, value_range=(-1, 1))

            plt.figure(figsize=(10, 4))

            # Display based on channel configuration
            # make_grid returns C, H, W. imshow expects H, W, C for color, H, W for grayscale.
            if IMG_CH == 1:
                plt.imshow(grid.squeeze(0).cpu().numpy(), cmap='gray') # Squeeze channel dim for grayscale
            else:
                plt.imshow(grid.permute(1, 2, 0).cpu().numpy()) # Permute to H,W,C for color

            plt.axis('off')
            plt.title('Generated Samples')
            plt.show()
        else:
            print("No samples generated.")
    model.train() # Set model back to training mode


# Define helper functions for safely saving models
def safe_save_model(model, path, optimizer=None, epoch=None, best_loss=None):
    """
    Safely saves model with error handling and backup.
    """
    try:
        # Create a dictionary with all the elements to save
        save_dict = {
            'model_state_dict': model.state_dict(),
        }

        # Add optional elements if provided
        if optimizer is not None:
            save_dict['optimizer_state_dict'] = optimizer.state_dict()
        if epoch is not None:
            save_dict['epoch'] = epoch
        if best_loss is not None:
            save_dict['best_loss'] = best_loss

        # Create a backup of previous checkpoint if it exists
        if os.path.exists(path):
            backup_path = path + '.backup'
            try:
                os.replace(path, backup_path)
                print(f"Created backup at {backup_path}")
            except Exception as e:
                print(f"Warning: Could not create backup - {e}")

        # Save the new checkpoint
        torch.save(save_dict, path)
        print(f"Model successfully saved to {path}")

    except Exception as e:
        print(f"Error saving model: {e}")
        print("Attempting emergency save...")

        try:
            emergency_path = path + '.emergency'
            # Save only the model state_dict for an emergency backup
            torch.save(model.state_dict(), emergency_path)
            print(f"Emergency save successful: {emergency_path}")
        except Exception as ee: # Catch specific emergency save error
            print(f"Emergency save failed: {ee}. Could not save model.")
```

```
#  Implementation of the training step function
def train_step(x, c):
    """
    Performs a single training step for the diffusion model.

    This function:
    1. Prepares class conditioning
    2. Samples random timesteps for each image
    3. Adds corresponding noise to the images
    4. Asks the model to predict the noise
    5. Calculates the loss between predicted and actual noise

    Args:
        x (torch.Tensor): Batch of clean images [batch_size, channels, height, width]
        c (torch.Tensor): Batch of class labels [batch_size]

    Returns:
        torch.Tensor: Mean squared error loss value
    """
    # Convert number labels to one-hot encoding for class conditioning
    # Example: Label 3 -> [0, 0, 0, 1, 0, 0, 0, 0, 0, 0] for MNIST
    c_one_hot = F.one_hot(c, N_CLASSES).float().to(device)

    # Create conditioning mask (all ones for standard training)
    # This would be used for classifier-free guidance if implemented
    c_mask = torch.ones_like(c.unsqueeze(-1)).to(device)

    # Pick random timesteps for each image in the batch
    # Different timesteps allow the model to learn the entire diffusion process
    t = torch.randint(0, n_steps, (x.shape[0],)).to(device)

    # Add noise to images according to the forward diffusion process
    # This simulates images at different stages of the diffusion process
    # Hint: Use the add_noise function you defined earlier

    # Enter your code here:

    x_t, noise = add_noise(x, t) # x_t is the noisy image, noise is the actual noise added

    # The model tries to predict the exact noise that was added
    # This is the core learning objective of diffusion models
    predicted_noise = model(x_t, t, c_one_hot, c_mask)

    # Calculate loss: how accurately did the model predict the noise?
    # MSE loss works well for image-based diffusion models
    # Hint: Use F.mse_loss to compare predicted and actual noise

    # Enter your code here:
    loss = F.mse_loss(predicted_noise, noise) # Compare the model's prediction with the ground truth noise


# Implementation of the main training loop
# Training configuration
early_stopping_patience = 10  # Number of epochs without improvement before stopping
gradient_clip_value = 1.0     # Maximum gradient norm for stability
display_frequency = 100       # How often to show progress (in steps)
generate_frequency = 500      # How often to generate samples (in steps)

# Progress tracking variables
best_loss = float('inf')
train_losses = []
val_losses = []
no_improve_epochs = 0

# Initialize the model, optimizer, and scheduler
# Define model parameters (adjust down_chs, t_embed_dim, c_embed_dim based on dataset and resources)
# For MNIST, these parameters are a reasonable starting point
model_params = {
    'T': n_steps, # Total diffusion steps
    'img_ch': IMG_CH,
    'img_size': IMG_SIZE,
    'down_chs': [64, 128, 256], # Channel dimensions for U-Net levels
    't_embed_dim': 256,          # Dimension for time embeddings
    'c_embed_dim': 256     # Dimension for class embeddings (usually N_CLASSES)
}


model = UNet(**model_params).to(device)
```

```
# Define the optimizer (Adam is a good choice for diffusion models)
optimizer = Adam(model.parameters(), lr=1e-4) # Learning rate can be tuned

# Define the learning rate scheduler
# ReduceLROnPlateau is a common choice; it reduces LR when validation loss stops improving
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer,
    mode='min', # Monitor validation loss (minimize)
    factor=0.5, # Reduce LR by half
    patience=5, # Wait for 5 epochs without improvement before reducing
    verbose=True # Print messages when LR is reduced
)


# Optional: Print model summary and parameter count
print("Model Architecture:")
print(model)
validate_model_parameters(model)


# Training loop
print("\n" + "="*50)
print("STARTING TRAINING")
print("="*50)
model.train()

# Wrap the training loop in a try-except block for better error handling:
# Your code for the training loop
# Hint: Use a try-except block for better error handling
# Process each epoch and each batch, with validation after each epoch

# Enter your code here:

try:
    for epoch in range(EPOCHS):
        print(f"\nEpoch {epoch+1}/{EPOCHS}")
        print("-" * 20)

        # Training phase
        model.train()
        epoch_losses = []

        # Process each batch
        for step, (images, labels) in enumerate(train_dataloader):  # Fixed: dataloader → train_dataloader
            images = images.to(device)
            labels = labels.to(device)

            # Training step
            optimizer.zero_grad()
            loss = train_step(images, labels)
            loss.backward()

            # Add gradient clipping for stability
            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=gradient_clip_value)

            optimizer.step()
            epoch_losses.append(loss.item())

            # Show progress at regular intervals
            if (step + 1) % display_frequency == 0:
                print(f"  Step {step + 1}/{len(train_dataloader)}, Loss: {loss.item():.4f}")

                # Generate samples less frequently to save time
                # This generates samples during the training phase, not just at epoch end.
                if (step + 1) % generate_frequency == 0 and step > 0:
                    print("  Generating samples during training...")
                    generate_samples(model, n_samples=5)

        # End of epoch - calculate average training loss
        if epoch_losses: # Ensure epoch_losses is not empty to avoid division by zero
            avg_train_loss = sum(epoch_losses) / len(epoch_losses)
            train_losses.append(avg_train_loss)
            print(f"\nTraining - Epoch {epoch+1} average loss: {avg_train_loss:.4f}")
        else:
            avg_train_loss = float('inf') # Or handle as per your logic
            print(f"\nTraining - Epoch {epoch+1}: No batches processed. Average loss: N/A")
```

```
        # Validation phase
        model.eval()
        val_epoch_losses = []
        print("Running validation...")

        with torch.no_grad():  # Disable gradients for validation
            for val_images, val_labels in val_dataloader:
                val_images = val_images.to(device)
                val_labels = val_labels.to(device)

                # Calculate validation loss
                val_loss = train_step(val_images, val_labels)
                val_epoch_losses.append(val_loss.item())

        # Calculate average validation loss
        if val_epoch_losses: # Ensure val_epoch_losses is not empty
            avg_val_loss = sum(val_epoch_losses) / len(val_epoch_losses)
            val_losses.append(avg_val_loss)
            print(f"Validation - Epoch {epoch+1} average loss: {avg_val_loss:.4f}")
        else:
            avg_val_loss = float('inf') # Or handle as per your logic
            print(f"Validation - Epoch {epoch+1}: No validation batches processed. Average loss: N/A")


        # Learning rate scheduling based on validation loss
        scheduler.step(avg_val_loss)
        current_lr = optimizer.param_groups[0]['lr']
        print(f"Learning rate: {current_lr:.6f}")

         # Generate samples at the end of each epoch (or every other, etc.)
        # This is typically for a more complete visual check than during training steps.
        if (epoch + 1) % 2 == 0 or epoch == EPOCHS - 1: # Adjusted for 1-based epoch check
            print("\nGenerating samples for visual progress check (end of epoch)...")
            generate_samples(model, n_samples=min(10, N_CLASSES)) # Generate up to N_CLASSES samples

        # Save best model based on validation loss
        if avg_val_loss < best_loss:
            best_loss = avg_val_loss
            # Use safe_save_model instead of just saving state_dict
            safe_save_model(model, 'best_diffusion_model.pt', optimizer, epoch + 1, best_loss)
            print(f"✓ New best model saved! (Val Loss: {best_loss:.4f})")
            no_improve_epochs = 0
        else:
            no_improve_epochs += 1
            print(f"No improvement for {no_improve_epochs}/{early_stopping_patience} epochs")

        # Early stopping
        if no_improve_epochs >= early_stopping_patience:
            print("\nEarly stopping triggered! No improvement in validation loss.")
            break

        # Plot loss curves every few epochs
        if (epoch  + 1) % 5 == 0 or epoch == EPOCHS - 1:
            plt.figure(figsize=(10, 5))
            plt.plot(train_losses, label='Training Loss')
            plt.plot(val_losses, label='Validation Loss')
            plt.xlabel('Epoch')
            plt.ylabel('Loss')
            plt.title('Training and Validation Loss')
            plt.legend()
            plt.grid(True)
            plt.show()

except KeyboardInterrupt:
    print("\nTraining interrupted by user. Performing graceful shutdown...")
except Exception as e:
    print(f"\nAn unexpected error occurred during training: {e}")
    import traceback
    traceback.print_exc() # Print full traceback for debugging



# Final wrap-up
print("\n" + "="*50)
print("TRAINING COMPLETE"if 'epoch' in locals() and epoch < EPOCHS -1 else "TRAINING FINISHED") # Check if training completed or broke early)
print("="*50)
print(f"Best validation loss: {best_loss:.4f}")
```

```
    print(f"Best validation loss: {best_loss:.4f}")

    # Generate final samples
    if 'model' in locals(): # Ensure model exists before trying to use it
        print("Generating final samples...")
        # Ensure model is in eval mode before final sample generation
        model.eval()
        generate_samples(model, n_samples=min(10, N_CLASSES))

    # Display final loss curves
    if train_losses or val_losses: # Only plot if there's data
            plt.figure(figsize=(12, 5))
            plt.plot(train_losses, label='Training Loss')
            plt.plot(val_losses, label='Validation Loss')
            plt.xlabel('Epoch')
            plt.ylabel('Loss')
            plt.title('Training and Validation Loss')
            plt.legend()
            plt.grid(True)
            plt.show()
    else:
        print("No loss data to plot.")


    # Clean up memory
    print("Cleaning up CUDA memory...")
    torch.cuda.empty_cache()
    print("Cleanup complete.")
```

```
Created DownBlock: in_chs=128, out_chs=256, spatial_reduction=2x
Created UpBlock: in_chs=256, skip_chs=128, out_chs=128, spatial_increase=2x
Created UpBlock: in_chs=128, skip_chs=64, out_chs=64, spatial_increase=2x
Created UNet with 3 scale levels
Channel dimensions: [64, 128, 256]
Time embedding dim: 256, Class embedding dim: 256
Model Architecture:
UNet(
  (time_embed): Sequential(
    (0): SinusoidalPositionEmbedBlock()
    (1): Linear(in_features=256, out_features=256, bias=True)
    (2): GELU(approximate='none')
  )
  (class_embed): EmbedBlock(
    (model): Sequential(
      (0): Linear(in_features=10, out_features=256, bias=True)
      (1): GELU(approximate='none')
      (2): Linear(in_features=256, out_features=256, bias=True)
      (3): GELU(approximate='none')
      (4): Unflatten(dim=1, unflattened_size=(256, 1, 1))
    )
  )
  (init_conv): GELUConvBlock(
    (block): Sequential(
      (0): Conv2d(1, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): GroupNorm(8, 64, eps=1e-05, affine=True)
      (2): GELU(approximate='none')
    )
  )
  (downs): ModuleList(
    (0): DownBlock(
      (model): Sequential(
        (0): GELUConvBlock(
          (block): Sequential(
            (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (1): GroupNorm(8, 128, eps=1e-05, affine=True)
            (2): GELU(approximate='none')
          )
        )
        (1): GELUConvBlock(
          (block): Sequential(
            (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (1): GroupNorm(8, 128, eps=1e-05, affine=True)
            (2): GELU(approximate='none')
          )
        )
        (2): RearrangePoolBlock(
          (rearrange): Rearrange('b c (h p1) (w p2) -> b (c p1 p2) h w', p1=2, p2=2)
          (conv_block): GELUConvBlock(
            (block): Sequential(
              (0): Conv2d(512, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
              (1): GroupNorm(8, 128, eps=1e-05, affine=True)
              (2): GELU(approximate='none')
            )
          )
        )
      )
    )
    (1): DownBlock(
      (model): Sequential(
        (0): GELUConvBlock(
          (block): Sequential(
            (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (1): GroupNorm(8, 256, eps=1e-05, affine=True)
            (2): GELU(approximate='none')
          )
        )
        (1): GELUConvBlock(
          (block): Sequential(
            (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (1): GroupNorm(8, 256, eps=1e-05, affine=True)
            (2): GELU(approximate='none')
          )
        )
        (2): RearrangePoolBlock(
          (rearrange): Rearrange('b c (h p1) (w p2) -> b (c p1 p2) h w', p1=2, p2=2)
          (conv_block): GELUConvBlock(
            (block): Sequential(
              (0): Conv2d(1024, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
              (1): GroupNorm(8, 256, eps=1e-05, affine=True)
              (2): GELU(approximate='none')
            )
          )
```

```
          )
        )
      )
    )
    (mid): Sequential(
      (0): GELUConvBlock(
        (block): Sequential(
          (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (1): GroupNorm(8, 512, eps=1e-05, affine=True)
          (2): GELU(approximate='none')
        )
      )
      (1): GELUConvBlock(
        (block): Sequential(
          (0): Conv2d(512, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (1): GroupNorm(8, 256, eps=1e-05, affine=True)
          (2): GELU(approximate='none')
        )
      )
    )
    (ups): ModuleList(
      (0): UpBlock(
        (upsample): ConvTranspose2d(256, 128, kernel_size=(2, 2), stride=(2, 2))
        (conv_blocks): Sequential(
          (0): GELUConvBlock(
            (block): Sequential(
              (0): Conv2d(384, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
              (1): GroupNorm(8, 128, eps=1e-05, affine=True)
              (2): GELU(approximate='none')
            )
          )
          (1): GELUConvBlock(
            (block): Sequential(
              (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
              (1): GroupNorm(8, 128, eps=1e-05, affine=True)
              (2): GELU(approximate='none')
            )
          )
        )
      )
      (1): UpBlock(
        (upsample): ConvTranspose2d(128, 64, kernel_size=(2, 2), stride=(2, 2))
        (conv_blocks): Sequential(
          (0): GELUConvBlock(
            (block): Sequential(
              (0): Conv2d(192, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
              (1): GroupNorm(8, 64, eps=1e-05, affine=True)
              (2): GELU(approximate='none')
            )
          )
          (1): GELUConvBlock(
            (block): Sequential(
              (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
              (1): GroupNorm(8, 64, eps=1e-05, affine=True)
              (2): GELU(approximate='none')
            )
          )
        )
      )
    )
    (final_conv): Conv2d(64, 1, kernel_size=(1, 1), stride=(1, 1))
  )
Total parameters: 7,457,793
Trainable parameters: 7,457,793
Estimated model memory (parameters + gradients + optimizer states, float32/float16): 113.8 MB

==================================================
STARTING TRAINING
==================================================

Epoch 1/30
--------------------
/usr/local/lib/python3.11/dist-packages/torch/optim/lr_scheduler.py:62: UserWarning: The verbose parameter is deprecated. Please use
  warnings.warn(

An unexpected error occurred during training: 'UNet' object has no attribute 'final_upsample'

==================================================
TRAINING COMPLETE
==================================================
Best validation loss: inf
Generating final samples...
Traceback (most recent call last):
  File "/tmp/ipython-input-22-3003682458.py", line 78, in <cell line: 0>
    loss = train_step(images, labels)
```

```
          ^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/tmp/ipython-input-21-3276455155.py", line 42, in train_step
    predicted_noise = model(x_t, t, c_one_hot, c_mask)
                      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py", line 1739, in _wrapped_call_impl
    return self._call_impl(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py", line 1750, in _call_impl
    return forward_call(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/tmp/ipython-input-19-1981487582.py", line 178, in forward
    x = self.final_upsample(x)
        ^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py", line 1928, in __getattr__
    raise AttributeError(
AttributeError: 'UNet' object has no attribute 'final_upsample'
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/tmp/ipython-input-22-3003682458.py in <cell line: 0>()
    187     # Ensure model is in eval mode before final sample generation
    188     model.eval()
--> 189     generate_samples(model, n_samples=min(10, N_CLASSES))
    190 
    191 # Display final loss curves
```

6 frames

```
/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py in __getattr__(self, name)
   1926             if name in modules:
   1927                 return modules[name]
-> 1928         raise AttributeError(
   1929             f"'{type(self).__name__}' object has no attribute '{name}'"
   1930         )
```

Next steps: Explain error

```
AttributeError: 'UNet' object has no attribute 'final_upsample'
```

```
# Plot training progress
plt.figure(figsize=(12, 5))

# Plot training and validation losses for comparison
plt.plot(train_losses, label='Training Loss')
if len(val_losses) > 0:  # Only plot validation if it exists
    plt.plot(val_losses, label='Validation Loss')

# Improve the plot with better labels and styling
plt.title('Diffusion Model Training Progress')
plt.xlabel('Epoch')
plt.ylabel('Loss (MSE)')
plt.legend()
plt.grid(True)

# Add annotations for key points
if len(train_losses) > 1:
    min_train_idx = train_losses.index(min(train_losses))
    plt.annotate(f'Min: {min(train_losses):.4f}',
                 xy=(min_train_idx, min(train_losses)),
                 xytext=(min_train_idx, min(train_losses)*1.2),
                 arrowprops=dict(facecolor='black', shrink=0.05),
                 fontsize=9)

# Add validation min point if available
if len(val_losses) > 1:
    min_val_idx = val_losses.index(min(val_losses))
    plt.annotate(f'Min: {min(val_losses):.4f}',
                xy=(min_val_idx, min(val_losses)),
                xytext=(min_val_idx, min(val_losses)*0.8),
                arrowprops=dict(facecolor='black', shrink=0.05),
                fontsize=9)

# Set y-axis to start from 0 or slightly lower than min value
plt.ylim(bottom=max(0, min(min(train_losses) if train_losses else float('inf'),
                          min(val_losses) if val_losses else float('inf'))*0.9))

plt.tight_layout()
plt.show()

# Add statistics summary for students to analyze
print("\nTraining Statistics:")
print("-" * 30)
if train_losses:
    print(f"Starting training loss:    {train_losses[0]:.4f}")
    print(f"Final training loss:       {train_losses[-1]:.4f}")
    print(f"Best training loss:        {min(train_losses):.4f}")
    print(f"Training loss improvement: {((train_losses[0] - min(train_losses)) / train_losses[0] * 100):.1f}%")

if val_losses:
    print("\nValidation Statistics:")
    print("-" * 30)
    print(f"Starting validation loss: {val_losses[0]:.4f}")
    print(f"Final validation loss:    {val_losses[-1]:.4f}")
    print(f"Best validation loss:     {min(val_losses):.4f}")

# STUDENT EXERCISE:
# 1. Try modifying this plot to show a smoothed version of the losses
# 2. Create a second plot showing the ratio of validation to training loss
#    (which can indicate overfitting when the ratio increases)
```

Step 6: Generating New Images

Now that our model is trained, let's generate some new images! We can:

1. Generate specific numbers
2. Generate multiple versions of each number
3. See how the generation process works step by step

```
def generate_number(model, number, n_samples=4):
    """
    Generate multiple versions of a specific number using the diffusion model.
```

```
    Args:
        model (nn.Module): The trained diffusion model
        number (int): The digit to generate (0-9)
        n_samples (int): Number of variations to generate

    Returns:
        torch.Tensor: Generated images of shape [n_samples, IMG_CH, IMG_SIZE, IMG_SIZE]
    """
    model.eval()  # Set model to evaluation mode
    with torch.no_grad():  # No need for gradients during generation
        # Start with random noise
        samples = torch.randn(n_samples, IMG_CH, IMG_SIZE, IMG_SIZE).to(device)

        # Set up the number we want to generate
        c = torch.full((n_samples,), number).to(device)
        c_one_hot = F.one_hot(c, N_CLASSES).float().to(device)
        # Correctly sized conditioning mask
        c_mask = torch.ones_like(c.unsqueeze(-1)).to(device)

        # Display progress information
        print(f"Generating {n_samples} versions of number {number}...")

        # Remove noise step by step
        for t in range(n_steps-1, -1, -1):
            t_batch = torch.full((n_samples,), t).to(device)
            samples = remove_noise(samples, t_batch, model, c_one_hot, c_mask)

            # Optional: Display occasional progress updates
            if t % (n_steps // 5) == 0:
                print(f"  Denoising step {n_steps-1-t}/{n_steps-1} completed")

        return samples

# Generate 4 versions of each number
plt.figure(figsize=(20, 10))
for i in range(10):
    # Generate samples for current digit
    samples = generate_number(model, i, n_samples=4)

    # Display each sample
    for j in range(4):
        # Use 2 rows, 10 digits per row, 4 samples per digit
        # i//5 determines the row (0 or 1)
        # i%5 determines the position in the row (0-4)
        # j is the sample index within each digit (0-3)
        plt.subplot(5, 8, (i%5)*8 + (i//5)*4 + j + 1)

        # Display the image correctly based on channel configuration
        if IMG_CH == 1:  # Grayscale
            plt.imshow(samples[j][0].cpu(), cmap='gray')
        else:  # Color image
            img = samples[j].permute(1, 2, 0).cpu()
            # Rescale from [-1, 1] to [0, 1] if needed
            if img.min() < 0:
                img = (img + 1) / 2
            plt.imshow(img)

        plt.title(f'Digit {i}')
        plt.axis('off')

plt.tight_layout()
plt.show()

# STUDENT ACTIVITY: Try generating the same digit with different noise seeds
# This shows the variety of styles the model can produce
print("\nSTUDENT ACTIVITY: Generating numbers with different noise seeds")

# Helper function to generate with seed
def generate_with_seed(number, seed_value=42, n_samples=10):
    torch.manual_seed(seed_value)
    return generate_number(model, number, n_samples)

# Pick a image and show many variations
# Hint select a image e.g. dog  # Change this to any other in the dataset of subset you chose
# Hint 2 use variations = generate_with_seed
# Hint 3 use plt.figure and plt.imshow to display the variations
```

```
# Enter your code here:
```

Step 7: Watching the Generation Process

Let's see how our model turns random noise into clear images, step by step. This helps us understand how the diffusion process works!

```
def visualize_generation_steps(model, number, n_preview_steps=10):
    """
    Show how an image evolves from noise to a clear number
    """
    model.eval()
    with torch.no_grad():
        # Start with random noise
        x = torch.randn(1, IMG_CH, IMG_SIZE, IMG_SIZE).to(device)

        # Set up which number to generate
        c = torch.tensor([number]).to(device)
        c_one_hot = F.one_hot(c, N_CLASSES).float().to(device)
        c_mask = torch.ones_like(c_one_hot).to(device)

        # Calculate which steps to show
        steps_to_show = torch.linspace(n_steps-1, 0, n_preview_steps).long()

        # Store images for visualization
        images = []
        images.append(x[0].cpu())

        # Remove noise step by step
        for t in range(n_steps-1, -1, -1):
            t_batch = torch.full((1,), t).to(device)
            x = remove_noise(x, t_batch, model, c_one_hot, c_mask)

            if t in steps_to_show:
                images.append(x[0].cpu())

        # Show the progression
        plt.figure(figsize=(20, 3))
        for i, img in enumerate(images):
            plt.subplot(1, len(images), i+1)
            if IMG_CH == 1:
                plt.imshow(img[0], cmap='gray')
            else:
                img = img.permute(1, 2, 0)
                if img.min() < 0:
                    img = (img + 1) / 2
                plt.imshow(img)
            step = n_steps if i == 0 else steps_to_show[i-1]
            plt.title(f'Step {step}')
            plt.axis('off')
        plt.show()

# Show generation process for a few numbers
for number in [0, 3, 7]:
    print(f"\nGenerating number {number}:")
    visualize_generation_steps(model, number)
```

Step 8: Adding CLIP Evaluation

CLIP is a powerful AI model that can understand both images and text. We'll use it to:

1. Evaluate how realistic our generated images are
2. Score how well they match their intended numbers
3. Help guide the generation process towards better quality

```
## Step 8: Adding CLIP Evaluation

# CLIP (Contrastive Language-Image Pre-training) is a powerful model by OpenAI that connects text and images.
# We'll use it to evaluate how recognizable our generated digits are by measuring how strongly
# the CLIP model associates our generated images with text descriptions like "an image of the digit 7".
```

```
# First, we need to install CLIP and its dependencies
print("Setting up CLIP (Contrastive Language-Image Pre-training) model...")

# Track installation status
clip_available = False

try:
    # Install dependencies first - these help CLIP process text and images
    print("Installing CLIP dependencies...")
    !pip install -q ftfy regex tqdm

    # Install CLIP from GitHub
    print("Installing CLIP from GitHub repository...")
    !pip install -q git+https://github.com/openai/CLIP.git

    # Import and verify CLIP is working
    print("Importing CLIP...")
    import clip

    # Test that CLIP is functioning
    models = clip.available_models()
    print(f"✓ CLIP installation successful! Available models: {models}")
    clip_available = True

except ImportError:
    print("❌ Error importing CLIP. Installation might have failed.")
    print("Try manually running: !pip install git+https://github.com/openai/CLIP.git")
    print("If you're in a Colab notebook, try restarting the runtime after installation.")

except Exception as e:
    print(f"❌ Error during CLIP setup: {e}")
    print("Some CLIP functionality may not work correctly.")

# Provide guidance based on installation result
if clip_available:
    print("\nCLIP is now available for evaluating your generated images!")
else:
    print("\nWARNING: CLIP installation failed. We'll skip the CLIP evaluation parts.")

# Import necessary libraries
import functools
import torch.nn.functional as F
```

Below we are createing a helper function to manage GPU memory when using CLIP. CLIP can be memory-intensive, so this will help prevent out-of-memory errors:

```
# Memory management decorator to prevent GPU OOM errors
def manage_gpu_memory(func):
    """
    Decorator that ensures proper GPU memory management.

    This wraps functions that might use large amounts of GPU memory,
    making sure memory is properly freed after function execution.
    """
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        if torch.cuda.is_available():
            # Clear cache before running function
            torch.cuda.empty_cache()
            try:
                return func(*args, **kwargs)
            finally:
                # Clear cache after running function regardless of success/failure
                torch.cuda.empty_cache()
        return func(*args, **kwargs)
    return wrapper

#===============================================================================
# Step 8: CLIP Model Loading and Evaluation Setup
#===============================================================================
# CLIP (Contrastive Language-Image Pre-training) is a neural network that connects
# vision and language. It was trained on 400 million image-text pairs to understand
# the relationship between images and their descriptions.
```

```
# We use it here as an "evaluation judge" to assess our generated images.

# Load CLIP model with error handling
try:
    # Load the ViT-B/32 CLIP model (Vision Transformer-based)
    clip_model, clip_preprocess = clip.load("ViT-B/32", device=device)
    print(f"✓ Successfully loaded CLIP model: {clip_model.visual.__class__.__name__}")
except Exception as e:
    print(f"❌ Failed to load CLIP model: {e}")
    clip_available = False
    # Instead of raising an error, we'll continue with degraded functionality
    print("CLIP evaluation will be skipped. Generated images will still be displayed but without quality scores.")

def evaluate_with_clip(images, target_number, max_batch_size=16):
    """
    Use CLIP to evaluate generated images by measuring how well they match textual descriptions.

    This function acts like an "automatic critic" for our generated digits by measuring:
    1. How well they match the description of a handwritten digit
    2. How clear and well-formed they appear to be
    3. Whether they appear blurry or poorly formed

    The evaluation process works by:
    - Converting our images to a format CLIP understands
    - Creating text prompts that describe the qualities we want to measure
    - Computing similarity scores between images and these text descriptions
    - Returning normalized scores (probabilities) for each quality

    Args:
        images (torch.Tensor): Batch of generated images [batch_size, channels, height, width]
        target_number (int): The specific digit (0-9) the images should represent
        max_batch_size (int): Maximum images to process at once (prevents GPU out-of-memory errors)

    Returns:
        torch.Tensor: Similarity scores tensor of shape [batch_size, 3] with scores for:
                      [good handwritten digit, clear digit, blurry digit]
                      Each row sums to 1.0 (as probabilities)
    """
    # If CLIP isn't available, return placeholder scores
    if not clip_available:
        print("⚠️ CLIP not available. Returning default scores.")
        # Equal probabilities (0.33 for each category)
        return torch.ones(len(images), 3).to(device) / 3

    try:
        # For large batches, we process in chunks to avoid memory issues
        # This is crucial when working with big images or many samples
        if len(images) > max_batch_size:
            all_similarities = []

            # Process images in manageable chunks
            for i in range(0, len(images), max_batch_size):
                print(f"Processing CLIP batch {i//max_batch_size + 1}/{(len(images)-1)//max_batch_size + 1}")
                batch = images[i:i+max_batch_size]

                # Use context managers for efficiency and memory management:
                # - torch.no_grad(): disables gradient tracking (not needed for evaluation)
                # - torch.cuda.amp.autocast(): uses mixed precision to reduce memory usage
                with torch.no_grad(), torch.cuda.amp.autocast():
                    batch_similarities = _process_clip_batch(batch, target_number)
                    all_similarities.append(batch_similarities)

                # Explicitly free GPU memory between batches
                # This helps prevent cumulative memory buildup that could cause crashes
                torch.cuda.empty_cache()

            # Combine results from all batches into a single tensor
            return torch.cat(all_similarities, dim=0)
        else:
            # For small batches, process all at once
            with torch.no_grad(), torch.cuda.amp.autocast():
                return _process_clip_batch(images, target_number)

    except Exception as e:
        # If anything goes wrong, log the error but don't crash
        print(f"❌ Error in CLIP evaluation: {e}")
        print(f"Traceback: {traceback.format_exc()}")
```

```
        # Return default scores so the rest of the notebook can continue
        return torch.ones(len(images), 3).to(device) / 3

def _process_clip_batch(images, target_number):
    """
    Core CLIP processing function that computes similarity between images and text descriptions.

    This function handles the technical details of:
    1. Preparing relevant text prompts for evaluation
    2. Preprocessing images to CLIP's required format
    3. Extracting feature embeddings from both images and text
    4. Computing similarity scores between these embeddings

    The function includes advanced error handling for GPU memory issues,
    automatically reducing batch size if out-of-memory errors occur.

    Args:
        images (torch.Tensor): Batch of images to evaluate
        target_number (int): The digit these images should represent

    Returns:
        torch.Tensor: Normalized similarity scores between images and text descriptions
    """
    try:
        # Create text descriptions (prompts) to evaluate our generated digits
        # We check three distinct qualities:
        # 1. If it looks like a handwritten example of the target digit
        # 2. If it appears clear and well-formed
        # 3. If it appears blurry or poorly formed (negative case)
        text_inputs = torch.cat([
            clip.tokenize(f"A handwritten number {target_number}"),
            clip.tokenize(f"A clear, well-written digit {target_number}"),
            clip.tokenize(f"A blurry or unclear number")
        ]).to(device)

        # Process images for CLIP, which requires specific formatting:

        # 1. Handle different channel configurations (dataset-dependent)
        if IMG_CH == 1:
            # CLIP expects RGB images, so we repeat the grayscale channel 3 times
            # For example, MNIST/Fashion-MNIST are grayscale (1-channel)
            images_rgb = images.repeat(1, 3, 1, 1)
        else:
            # For RGB datasets like CIFAR-10/CelebA, we can use as-is
            images_rgb = images

        # 2. Normalize pixel values to [0,1] range if needed
        # Different datasets may have different normalization ranges
        if images_rgb.min() < 0:  # If normalized to [-1,1] range
            images_rgb = (images_rgb + 1) / 2  # Convert to [0,1] range

        # 3. Resize images to CLIP's expected input size (224x224 pixels)
        # CLIP was trained on this specific resolution
        resized_images = F.interpolate(images_rgb, size=(224, 224),
                                      mode='bilinear', align_corners=False)

        # Extract feature embeddings from both images and text prompts
        # These are high-dimensional vectors representing the content
        image_features = clip_model.encode_image(resized_images)
        text_features = clip_model.encode_text(text_inputs)

        # Normalize feature vectors to unit length (for cosine similarity)
        # This ensures we're measuring direction, not magnitude
        image_features = image_features / image_features.norm(dim=-1, keepdim=True)
        text_features = text_features / text_features.norm(dim=-1, keepdim=True)

        # Calculate similarity scores between image and text features
        # The matrix multiplication computes all pairwise dot products at once
        # Multiplying by 100 scales to percentage-like values before applying softmax
        similarity = (100.0 * image_features @ text_features.T).softmax(dim=-1)

        return similarity

    except RuntimeError as e:
        # Special handling for CUDA out-of-memory errors
        if "out of memory" in str(e):
            # Free GPU memory immediately
```