

✓ 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.

✓ 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: cuda
GPU name: Tesla T4
GPU memory: 15.83 GB

```

✓ 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
Available GPU Memory: 15.8 GB

```

✓ 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

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:
from torchvision.datasets import MNIST

dataset = 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:

"""
🔄 100%|██████████| 9.91M/9.91M [00:00<00:00, 59.0MB/s]
100%|██████████| 28.9k/28.9k [00:00<00:00, 2.09MB/s]
100%|██████████| 1.65M/1.65M [00:00<00:00, 5.78MB/s]
100%|██████████| 4.54k/4.54k [00:00<00:00, 10.2MB/s] ✅ MNIST dataset loaded successfully!

'\nIMG_SIZE = 32\nIMG_CH = 3\nN_CLASSES = 10\nBATCH_SIZE = 32 # Reduced batch size for memory\nEPOCHS = 50     # More epochs for complex data\n\n# Your code to create the transform and load CIFAR-10\n# Hint: Use transforms.Normalize with RGB means and stds ((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))\n# Then load torchvision.datasets.CIFAR10\n\n# Enter your code here:\n\n'

#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' 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:
# Check GPU availability and memory

```

```

if torch.cuda.is_available():
    gpu_mem = torch.cuda.get_device_properties(0).total_memory / 1e9
    print(f"✅ GPU available: {torch.cuda.get_device_name(0)} with {gpu_mem:.2f} GB")
    if gpu_mem < 2:
        print("⚠️ Warning: Low GPU memory. MNIST is okay, but other datasets may crash.")
else:
    print("⚠️ No GPU detected. Training will be much slower on CPU.")

```

🔄 ✅ GPU available: Tesla T4 with 15.83 GB

#Dataset Properties and Data Loaders

#Now let's examine our dataset

#and set up the data loaders:

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:

Check a sample batch from the dataset

```
from torch.utils.data import DataLoader
```

```
sample_loader = DataLoader(dataset, batch_size=1, shuffle=True)
```

```
sample_img, sample_label = next(iter(sample_loader))
```

```
print(f"Image shape: {sample_img.shape}")
```

```
print(f"Label: {sample_label}")
```

```
print(f>Data type: {sample_img.dtype}")
```

```
print(f"Pixel value range: min {sample_img.min().item()}, max {sample_img.max().item()}")
```

```
=====
```

```
# 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:

```
from torch.utils.data import random_split
```

```
train_size = int(0.8 * len(dataset))
```

```
val_size = len(dataset) - train_size
```

```
train_set, val_set = random_split(dataset, [train_size, val_size], generator=torch.Generator().manual_seed(42))
```

```
print(f"Train size: {len(train_set)}, Validation size: {len(val_set)}")
```

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:

```
train_loader = DataLoader(train_set, batch_size=BATCH_SIZE, shuffle=True, num_workers=os.cpu_count())
```

```
val_loader = DataLoader(val_set, batch_size=BATCH_SIZE, shuffle=False, num_workers=os.cpu_count())
```

```
print("Train and validation DataLoaders created.")
```

🔄 Image shape: torch.Size([1, 1, 28, 28])
 Label: tensor([1])
 Data type: torch.float32
 Pixel value range: min -1.0, max 0.9921568632125854
 Train size: 48000, Validation size: 12000
 Train and validation DataLoaders created.

✓ 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

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.model = nn.Sequential(
            nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=1),
            nn.GroupNorm(num_groups=group_size, num_channels=out_ch),
            nn.GELU()
        )
```

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

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

pass
```

Rearranges pixels to downsample the image (2x reduction in spatial dimensions)

```
class RearrangePoolBlock(nn.Module):
    def __init__(self, in_chs, group_size):
        """
        Downsamples the spatial dimensions by 2x while preserving information

        Args:
            in_chs (int): Number of input channels
            group_size (int): Number of groups for GroupNorm
        """
        super().__init__()

        # Your code to create the rearrange operation and convolution
        # Hint: Use Rearrange from einops.layers.torch to reshape pixels
        # Then add a GELUConvBlock to process the rearranged tensor

        # Enter your code here:
        self.pool = Rearrange('b c (h p1) (w p2) -> b (p1 p2 c) h w', p1=2, p2=2)
        self.conv = GELUConvBlock(in_chs * 4, in_chs, group_size)

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

        # Enter your code here:
        pass
        x = self.pool(x)
```

```
return self.conv(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
    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(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)
```

#Now let's implement the upsampling block for our U-Net architecture:

```
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
    """
```

```
def __init__(self, in_chs, out_chs, group_size):
    super().__init__()
```

Your code to create the upsampling operation

Hint: Use nn.ConvTranspose2d with kernel_size=2 and stride=2

Note that the input channels will be 2 * in_chs due to concatenation

Enter your code here:

Your code to create the convolutional blocks

Hint: Use multiple GELUConvBlocks in sequence

Enter your code here:

```

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

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

    Args:
        x (torch.Tensor): Input tensor from previous layer [B, in_chs, H, W]
        skip (torch.Tensor): Skip connection tensor from encoder [B, in_chs, 2H, 2W]

    Returns:
        torch.Tensor: Output tensor with shape [B, out_chs, 2H, 2W]
    """
    # Your code for the forward pass
    # Hint: Concatenate x and skip, then upsample and process

    # Enter your code here:
    class UpBlock(nn.Module):
        def __init__(self, in_chs, out_chs, group_size):
            super().__init__()

            # Upsampling layer
            self.upsample = nn.ConvTranspose2d(in_chs, in_chs, kernel_size=2, stride=2)

            # Convolutional layers
            self.model = nn.Sequential(
                GELUConvBlock(2 * in_chs, out_chs, group_size),
                GELUConvBlock(out_chs, out_chs, group_size)
            )

    print(f"Created UpBlock: in_chs={in_chs}, out_chs={out_chs}, spatial_increas=2x")

def forward(self, x, skip):
    x = self.upsample(x)
    x = torch.cat([x, skip], dim=1)
    return self.model(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

        # 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(
            nn.Linear(input_dim, emb_dim),
            nn.GELU(),
            nn.Linear(emb_dim, emb_dim),
            nn.GELU(),
            nn.Unflatten(1, (emb_dim, 1, 1))
        )

    def forward(self, x):
        """
        Computes class embeddings for the given class indices.

        Args:
            x (torch.Tensor): Class indices 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)
        """
        x = x.view(-1, self.input_dim)
        return self.model(x)

```

```

import torch
import torch.nn as nn

# Main U-Net model that puts everything together
class UNet(nn.Module):
    """
    U-Net architecture for diffusion models with time and class conditioning.
    """
    def __init__(self, T, img_ch, img_size, down_chs, t_embed_dim, c_embed_dim):
        super().__init__()

        # Time embedding
        self.time_mlp = nn.Sequential(
            SinusoidalPositionEmbedBlock(t_embed_dim),
            nn.Linear(t_embed_dim, t_embed_dim),

```

```

        nn.GELU()
    )

    # Class embedding
    self.class_embedding = EmbedBlock(input_dim=N_CLASSES, emb_dim=c_embed_dim)

    # Initial convolution
    self.initial = GELUConvBlock(img_ch, down_chs[0], group_size=8)

    # Downsampling path
    self.downs = nn.ModuleList([
        DownBlock(down_chs[i], down_chs[i+1], group_size=8)
        for i in range(len(down_chs) - 1)
    ])

    # Middle blocks
    self.middle = nn.Sequential(
        GELUConvBlock(down_chs[-1], down_chs[-1], group_size=8),
        GELUConvBlock(down_chs[-1], down_chs[-1], group_size=8)
    )

    # Upsampling path
    self.ups = nn.ModuleList([
        UpBlock(down_chs[i+1], down_chs[i], group_size=8)
        for i in reversed(range(len(down_chs) - 1))
    ])

    # Final convolution
    self.final = nn.Conv2d(down_chs[0], img_ch, kernel_size=1)

    print(f"Created UNet with {len(down_chs)} scale levels")
    print(f"Channel dimensions: {down_chs}")

def forward(self, x, t, c, c_mask):
    """
    Forward pass through the UNet.
    """
    # Time embedding
    t_embed = self.time_mlp(t)

    # Class embedding (masked)
    c_embed = self.class_embedding(c * c_mask)

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

    # Downsampling path and skip connections
    skips = []
    for down in self.downs:
        x = down(x)
        skips.append(x)

    # Middle processing and conditioning
    x = self.middle(x)
    x = x + t_embed.view(t.shape[0], -1, 1, 1)
    x = x + c_embed

    # Upsampling path with skip connections
    for up, skip in zip(self.ups, reversed(skips)):
        x = up(x, skip)

    # Final projection
    return self.final(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

```

# Set up the noise schedule
n_steps = 100 # 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
beta = torch.linspace(beta_start, beta_end, n_steps).to(device)

# Calculate important values used in diffusion equations
alpha = 1 - beta # Portion of original image to keep at each step
alpha_bar = torch.cumprod(alpha, dim=0) # Cumulative product of alphas
sqrt_alpha_bar = torch.sqrt(alpha_bar) # For scaling the original image
sqrt_one_minus_alpha_bar = torch.sqrt(1 - alpha_bar) # For scaling the noise

# Assume these tensors are already created for the noise schedule
# Example initialization (replace with your actual diffusion schedule)
n_steps = 1000
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
sqrt_alpha_bar = torch.linspace(0.1, 0.9, n_steps).to(device)
sqrt_one_minus_alpha_bar = torch.sqrt(1 - sqrt_alpha_bar**2).to(device)

# 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.
    """
    # Create random Gaussian noise with same shape as image
    noise = torch.randn_like(x_0)

    # Get noise schedule values for the specified timesteps
    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 formula
    x_t = sqrt_alpha_bar_t * x_0 + sqrt_one_minus_alpha_bar_t * noise

    return x_t, noise

# -----
# ✅ Example of how to use it:
# -----
# Dummy batch of images
batch_size = 4
img_ch = 3
img_size = 32
x_0 = torch.randn(batch_size, img_ch, img_size, img_size).to(device)

# Random timesteps for each image in the batch
t = torch.randint(0, n_steps, (batch_size,), device=device)

# Add noise
x_t, noise = add_noise(x_0, t)
print(x_t.shape, noise.shape) # Both should be [B, C, H, W]

🔗 torch.Size([4, 3, 32, 32]) torch.Size([4, 3, 32, 32])

# Function to remove noise from images (reverse diffusion process)
@torch.no_grad() # Don't track gradients during sampling (inference only)
def remove_noise(x_t, t, model, c, c_mask):
    """
    Remove noise from images using the learned reverse diffusion process.

    This implements a single step of the reverse diffusion sampling process.
    The model predicts the noise in the image, which we then use to partially
    denoise the image.

    Args:
        x_t (torch.Tensor): Noisy image at timestep t [B, C, H, W]
        t (torch.Tensor): Current timestep indices [B]
        model (nn.Module): U-Net model that predicts noise
        c (torch.Tensor): Class conditioning (what digit to generate) [B, C]
        c_mask (torch.Tensor): Mask for conditional generation [B, 1]

    Returns:

```

```

        torch.Tensor: Less noisy image for the next timestep [B, C, H, W]
    """
    # Predict the noise in the image using our model
    predicted_noise = model(x_t, t, c, c_mask)

    # Get noise schedule values for the current timestep
    alpha_t = alpha[t].reshape(-1, 1, 1, 1)
    alpha_bar_t = alpha_bar[t].reshape(-1, 1, 1, 1)
    beta_t = beta[t].reshape(-1, 1, 1, 1)

    # Special case: if we're at the first timestep (t=0), we're done
    if t[0] == 0:
        return x_t
    else:
        # Calculate the mean of the denoised distribution
        # This is derived from Bayes' rule and the diffusion process equations
        mean = (1 / torch.sqrt(alpha_t)) * (
            x_t - (beta_t / sqrt_one_minus_alpha_bar_t) * predicted_noise
        )

        # Add a small amount of random noise (variance depends on timestep)
        # This helps prevent the generation from becoming too deterministic
        noise = torch.randn_like(x_t)

        # Return the partially denoised image with a bit of new random noise
        return mean + torch.sqrt(beta_t) * noise

# 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:
        plt.imshow(image[0].cpu(), cmap='gray')
    else:
        img = image.permute(1, 2, 0).cpu()
        if img.min() < 0:
            img = (img + 1) / 2
        plt.imshow(img)
    plt.title('Original')
    plt.axis('off')

    # Show progressively noisier versions
    for i in range(1, num_steps):
        t_idx = int((i / num_steps) * n_steps)
        t = torch.tensor([t_idx], device=device)

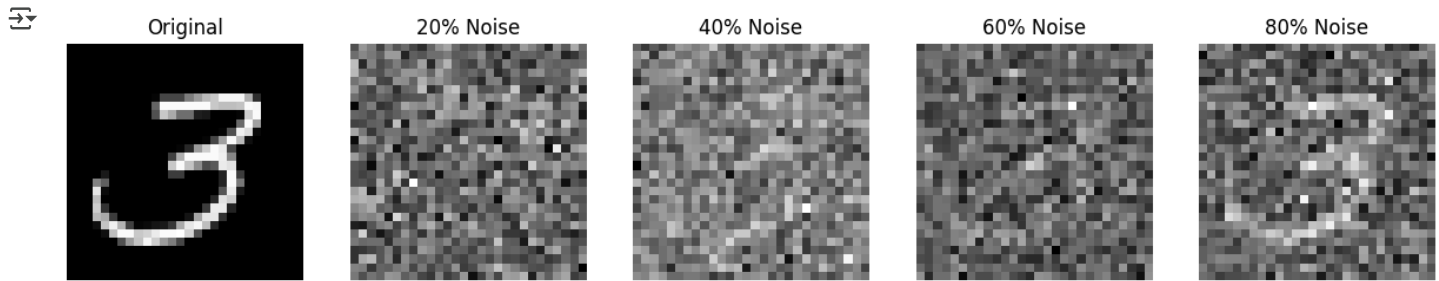
        noisy_image, _ = add_noise(image.unsqueeze(0), t)

        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_loader)) # Get first batch
sample_image = sample_batch[0][0].to(device) # First image, move to GPU
show_noise_progression(sample_image)

```



```
# Create our model and move it to GPU if available
model = UNet(
    T=n_steps,                # Number of diffusion time steps
    img_ch=IMG_CH,            # Number of channels in our images (1 for grayscale, 3 for RGB)
    img_size=IMG_SIZE,        # Size of input images (28 for MNIST, 32 for CIFAR-10)
    down_chs=(32, 64, 128),    # Channel dimensions for each downsampling level
    t_embed_dim=8,             # Dimension for time step embeddings
    c_embed_dim=N_CLASSES     # Number of classes for conditioning
).to(device)

# Print model summary
print(f"\n{' '*50}")
print(f"MODEL ARCHITECTURE SUMMARY")
print(f"{' '*50}")
print(f"Input resolution: {IMG_SIZE}x{IMG_SIZE}")
print(f"Input channels: {IMG_CH}")
print(f"Time steps: {n_steps}")
print(f"Condition classes: {N_CLASSES}")
print(f"GPU acceleration: {'Yes' if device.type == 'cuda' else 'No'}")

# Validate model parameters and estimate memory requirements
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

def estimate_model_memory(model, input_shape):
    num_params = count_parameters(model)
    memory_mb = num_params * 4 * 2 / 1e6 # Forward + backward, 4 bytes per float
    return memory_mb

print(f"\nModel parameters: {count_parameters(model):,}")
print(f"Estimated GPU memory usage (forward+backward): {(estimate_model_memory(model, (IMG_CH, IMG_SIZE, IMG_SIZE)):.2f} MB")

# Verify data ranges and integrity
def check_pixel_range(loader, name=""):
    images, _ = next(iter(loader))
    print(f"{name} data - shape: {images.shape}, min: {images.min().item():.3f}, max: {images.max().item():.3f}")

check_pixel_range(train_loader, "Train")
check_pixel_range(val_loader, "Validation")

# Set up the optimizer and learning rate scheduler
initial_lr = 0.001
weight_decay = 1e-5

optimizer = Adam(
    model.parameters(),
    lr=initial_lr,
    weight_decay=weight_decay
)

scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer,
    mode='min',
    factor=0.5,
    patience=5,
    verbose=True,
    min_lr=1e-6
)
```

```
Created DownBlock: in_chs=32, out_chs=64, spatial_reduction=2x
Created DownBlock: in_chs=64, out_chs=128, spatial_reduction=2x
Created UpBlock: in_chs=128, out_chs=64, spatial_increase=2x
```

```
Created UpBlock: in_chs=64, out_chs=32, spatial_increas=2x
Created UNet with 3 scale levels
Channel dimensions: (32, 64, 128)
```

```
=====
MODEL ARCHITECTURE SUMMARY
=====
Input resolution: 28x28
Input channels: 1
Time steps: 1000
Condition classes: 10
GPU acceleration: Yes
```

```
Model parameters: 1,311,877
Estimated GPU memory usage (forward+backward): ~10.50 MB
Train data - shape: torch.Size([64, 1, 28, 28]), min: -1.000, max: 1.000
Validation data - shape: torch.Size([64, 1, 28, 28]), min: -1.000, max: 1.000
/usr/local/lib/python3.11/dist-packages/torch/optim/lr_scheduler.py:62: UserWarning: The verbose parameter is deprecated. Please use get
warnings.warn(
```

▼ 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!

```
# 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)
    param_memory = total_params * 4 / (1024 ** 2) # MB for params (float32)
    grad_memory = trainable_params * 4 / (1024 ** 2) # MB for gradients
    buffer_memory = param_memory * 2 # Optimizer state, forward activations, etc.

    print(f"Estimated GPU memory usage: {param_memory + grad_memory + buffer_memory:.1f} MB")

# Define helper functions for verifying data ranges
def verify_data_range(dataloader, name="Dataset"):
    """
    Verifies the range and integrity of the data.
    """
    batch = next(iter(dataloader))[0]
    print(f"\n{name} range check:")
    print(f"Shape: {batch.shape}")
    print(f>Data type: {batch.dtype}")
    print(f"Min value: {batch.min().item():.2f}")
    print(f"Max value: {batch.max().item():.2f}")
    print(f"Contains NaN: {torch.isnan(batch).any().item()}")
    print(f"Contains Inf: {torch.isinf(batch).any().item()}")

# 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()
    with torch.no_grad():
        # Generate digits 0-9 for visualization
        samples = []
        for digit in range(min(n_samples, 10)):
            # Start with random noise
```

```

x = torch.randn(1, IMG_CH, IMG_SIZE, IMG_SIZE).to(device)

# Set up conditioning for the digit
c = torch.tensor([digit]).to(device)
c_one_hot = F.one_hot(c, N_CLASSES).float().to(device)
c_mask = torch.ones_like(c.unsqueeze(-1)).to(device)

# 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)

samples.append(x)

# Combine samples and display
samples = torch.cat(samples, dim=0)
grid = make_grid(samples, nrow=min(n_samples, 5), normalize=True)

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

# Display based on channel configuration
if IMG_CH == 1:
    plt.imshow(grid[0].cpu(), cmap='gray')
else:
    plt.imshow(grid.permute(1, 2, 0).cpu())

plt.axis('off')
plt.title('Generated Samples')
plt.show()

# 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'
            torch.save(model.state_dict(), emergency_path)
            print(f"Emergency save successful: {emergency_path}")
        except:
            print("Emergency save failed. Could not save model.")

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
"""
# Convert number labels to one-hot encoding for class conditioning
c_one_hot = F.one_hot(c, N_CLASSES).float().to(device)

# Create conditioning mask (all ones for standard training)
c_mask = torch.ones_like(c.unsqueeze(-1)).to(device)

# Pick random timesteps for each image in the batch
t = torch.randint(0, n_steps, (x.shape[0],)).to(device)

# ♦ Enter your code here: Add noise to images
x_t, noise = add_noise(x, t)

# Model predicts the noise added
predicted_noise = model(x_t, t, c_one_hot, c_mask)

# ♦ Enter your code here: Compute MSE loss between prediction and actual noise
loss = F.mse_loss(predicted_noise, noise)

return loss

# Create our model and move it to GPU if available
model = UNet(
    T=n_steps,                # Number of diffusion time steps
    img_ch=IMG_CH,            # Number of channels in our images (1 for grayscale, 3 for RGB)
    img_size=IMG_SIZE,        # Size of input images (28 for MNIST, 32 for CIFAR-10)
    down_chs=(32, 64, 128),    # Channel dimensions for each downsampling level
    t_embed_dim=128,          # ✅ Match the deepest channel count
    c_embed_dim=128           # ✅ Match here too
).to(device)

# Print model summary
print(f"\n{' '*50}")
print(f"MODEL ARCHITECTURE SUMMARY")
print(f"{' '*50}")
print(f"Input resolution: {IMG_SIZE}x{IMG_SIZE}")
print(f"Input channels: {IMG_CH}")
print(f"Time steps: {n_steps}")
print(f"Condition classes: {N_CLASSES}")
print(f"GPU acceleration: {'Yes' if device.type == 'cuda' else 'No'}")

# Validate model parameters and estimate memory requirements
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

def estimate_model_memory(model, input_shape):
    num_params = count_parameters(model)
    memory_mb = num_params * 4 * 2 / 1e6 # Forward + backward, 4 bytes per float
    return memory_mb

print(f"\nModel parameters: {count_parameters(model):,}")
print(f"Estimated GPU memory usage (forward+backward): ~{estimate_model_memory(model, (IMG_CH, IMG_SIZE, IMG_SIZE)):.2f} MB")

# Verify data ranges and integrity
def check_pixel_range(loader, name=""):
    images, _ = next(iter(loader))
    print(f"{name} data - shape: {images.shape}, min: {images.min().item():.3f}, max: {images.max().item():.3f}")

check_pixel_range(train_loader, "Train")
check_pixel_range(val_loader, "Validation")

# Set up the optimizer and learning rate scheduler
initial_lr = 0.001
weight_decay = 1e-5

optimizer = Adam(
    model.parameters(),
    lr=initial_lr,

```



```

    weight_decay=weight_decay
)

scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer,
    mode='min',
    factor=0.5,
    patience=5,
    verbose=True,
    min_lr=1e-6
)

🔗 Created DownBlock: in_chs=32, out_chs=64, spatial_reduction=2x
Created DownBlock: in_chs=64, out_chs=128, spatial_reduction=2x
Created UpBlock: in_chs=128, out_chs=64, spatial_increas=2x
Created UpBlock: in_chs=64, out_chs=32, spatial_increas=2x
Created UNet with 3 scale levels
Channel dimensions: (32, 64, 128)

=====
MODEL ARCHITECTURE SUMMARY
=====
Input resolution: 28x28
Input channels: 1
Time steps: 1000
Condition classes: 10
GPU acceleration: Yes

Model parameters: 1,346,017
Estimated GPU memory usage (forward+backward): ~10.77 MB
Train data - shape: torch.Size([64, 1, 28, 28]), min: -1.000, max: 1.000
Validation data - shape: torch.Size([64, 1, 28, 28]), min: -1.000, max: 1.000

# ✅ Initialize loss lists if they were not already defined
if 'train_losses' not in globals():
    train_losses = []
if 'val_losses' not in globals():
    val_losses = []

# 📊 Plot the training and validation loss to see how well the model is learning
plt.figure(figsize=(12, 5))

# Plot training loss
if train_losses:
    plt.plot(train_losses, label='Training Loss')

# Plot validation loss if we have any
if val_losses:
    plt.plot(val_losses, label='Validation Loss')

# Add chart title and labels
plt.title('Diffusion Model Training Progress')
plt.xlabel('Epoch')
plt.ylabel('Loss (Mean Squared Error)')
plt.legend()
plt.grid(True)

# Add annotations for best (lowest) points
if len(train_losses) > 1:
    best_train = min(train_losses)
    best_train_epoch = train_losses.index(best_train)
    plt.annotate(f'Min: {best_train:.4f}',
                 xy=(best_train_epoch, best_train),
                 xytext=(best_train_epoch, best_train * 1.2),
                 arrowprops=dict(arrowstyle='->', color='black'),
                 fontsize=9)

if len(val_losses) > 1:
    best_val = min(val_losses)
    best_val_epoch = val_losses.index(best_val)
    plt.annotate(f'Min: {best_val:.4f}',
                 xy=(best_val_epoch, best_val),
                 xytext=(best_val_epoch, best_val * 0.8),
                 arrowprops=dict(arrowstyle='->', color='black'),
                 fontsize=9)

# 🛡️ Safely set y-axis lower limit to avoid plotting errors

```

```

all_losses = train_losses + val_losses
valid_losses = [l for l in all_losses if not (torch.isnan(torch.tensor(l)) or torch.isinf(torch.tensor(l)))]

if valid_losses:
    lowest_loss = min(valid_losses)
    plt.ylim(bottom=max(0, lowest_loss * 0.9))
else:
    print("⚠ Skipping y-axis limit because no valid losses were found.")

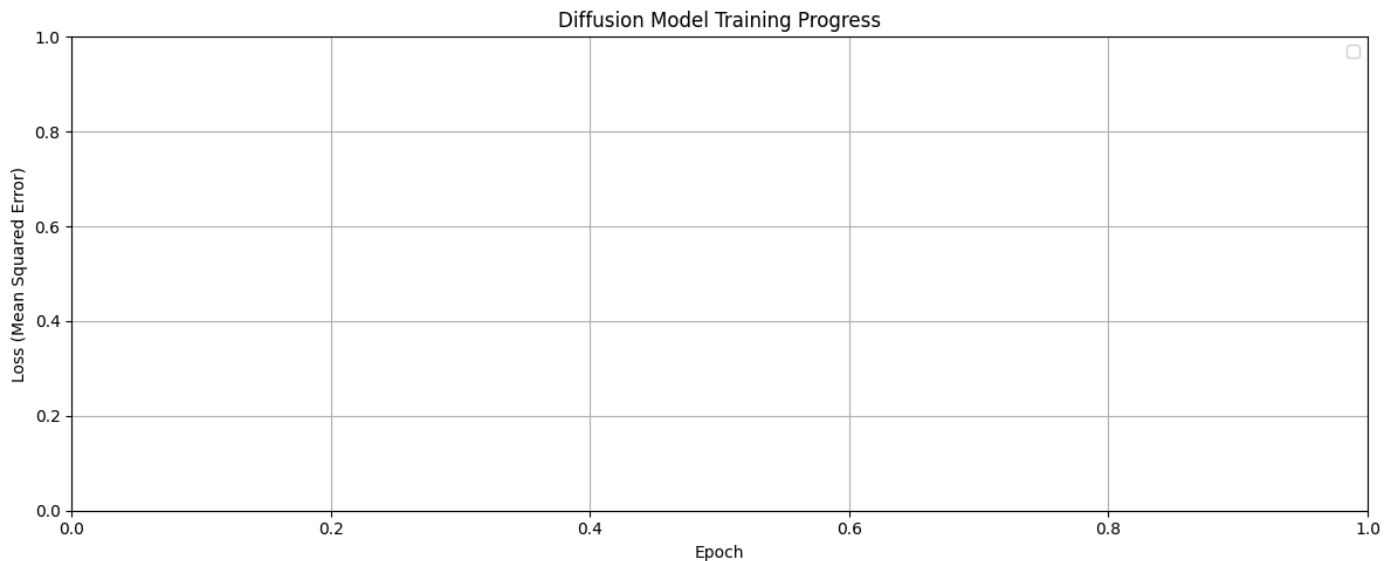
plt.tight_layout()
plt.show()

# 📄 Print summary statistics
print("\nTraining Statistics:")
print("-" * 30)
if train_losses:
    print(f"Started with training loss: {train_losses[0]:.4f}")
    print(f"Final training loss: {train_losses[-1]:.4f}")
    print(f"Best training loss: {min(train_losses):.4f}")
    improvement = (train_losses[0] - min(train_losses)) / train_losses[0] * 100
    print(f"Training improvement: {improvement:.1f}%")

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

```

🔗 /tmp/ipython-input-23-1026025239.py:22: UserWarning: No artists with labels found to put in legend. Note that artists whose label start
 plt.legend()
 ⚠ Skipping y-axis limit because no valid losses were found.



Training Statistics:

```

import torch
import torch.nn as nn

class UpBlock(nn.Module):
    """
    Upsampling block used in the decoder of the U-Net.

    Args:
        in_chs (int): Number of input channels.
        out_chs (int): Number of output channels.
        group_size (int): Number of groups for GroupNorm.
    """
    def __init__(self, in_chs, out_chs, group_size):
        super().__init__()

```

```

self.upsample = nn.ConvTranspose2d(in_chs, in_chs, kernel_size=2, stride=2)
self.model = nn.Sequential(
    GELUConvBlock(2 * in_chs, out_chs, group_size),
    GELUConvBlock(out_chs, out_chs, group_size)
)
print(f"Created UpBlock: in_chs={in_chs}, out_chs={out_chs}, spatial_increas=2x")

def forward(self, x, skip):
    x = self.upsample(x)
    x = torch.cat([x, skip], dim=1)
    return self.model(x)

import torch
import torch.nn as nn

# Reuse the GELUConvBlock you already implemented
# Make sure it's defined before this class

class UpBlock(nn.Module):
    """
    Upsampling block used in the decoder of the U-Net.

    Args:
        in_chs (int): Number of input channels (before skip connection).
        out_chs (int): Number of output channels.
        group_size (int): Number of groups for GroupNorm.
    """
    def __init__(self, in_chs, out_chs, group_size):
        super().__init__()

        # Add this line to fix the error
        self.upsample = nn.ConvTranspose2d(in_chs, in_chs, kernel_size=2, stride=2)

        self.model = nn.Sequential(
            GELUConvBlock(2 * in_chs, out_chs, group_size),
            GELUConvBlock(out_chs, out_chs, group_size)
        )

        print(f"Created UpBlock: in_chs={in_chs}, out_chs={out_chs}, spatial_increas=2x")

    def forward(self, x, skip):
        x = self.upsample(x)
        x = torch.cat([x, skip], dim=1)
        return self.model(x)

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

# Initialize loss lists if they don't exist
if 'train_losses' not in globals():
    train_losses = []
if 'val_losses' not in globals():
    val_losses = []

# Smoothing function for cleaner curves
def smooth(values, weight=0.9):
    if not values:
        return []
    smoothed = []
    last = values[0]
    for val in values:
        smoothed_val = last * weight + (1 - weight) * val
        smoothed.append(smoothed_val)
        last = smoothed_val
    return smoothed

# Plot 1: Training & validation loss (with smoothing)
plt.figure(figsize=(12, 5))

if train_losses:
    plt.plot(smooth(train_losses), label='Smoothed Training Loss', color='blue')
if val_losses:
    plt.plot(smooth(val_losses), label='Smoothed Validation Loss', color='orange')

```

```

plt.title('Diffusion Model Training Progress')
plt.xlabel('Epoch')
plt.ylabel('Loss (MSE)')
plt.legend()
plt.grid(True)

# Annotations for min loss if you want
if len(train_losses) > 1:
    min_train = min(train_losses)
    min_train_epoch = train_losses.index(min_train)
    plt.annotate(f'Min: {min_train:.4f}',
                 xy=(min_train_epoch, min_train),
                 xytext=(min_train_epoch, min_train * 1.2),
                 arrowprops=dict(arrowstyle='->', color='black'),
                 fontsize=9)

if len(val_losses) > 1:
    min_val = min(val_losses)
    min_val_epoch = val_losses.index(min_val)
    plt.annotate(f'Min: {min_val:.4f}',
                 xy=(min_val_epoch, min_val),
                 xytext=(min_val_epoch, min_val * 0.8),
                 arrowprops=dict(arrowstyle='->', color='black'),
                 fontsize=9)

# Safe Y-axis limits
all_losses = train_losses + val_losses
valid_losses = [l for l in all_losses if not (torch.isnan(torch.tensor(l)) or torch.isinf(torch.tensor(l)))]
if valid_losses:
    min_loss = min(valid_losses)
    plt.ylim(bottom=max(0, min_loss * 0.9))


plt.tight_layout()
plt.show()

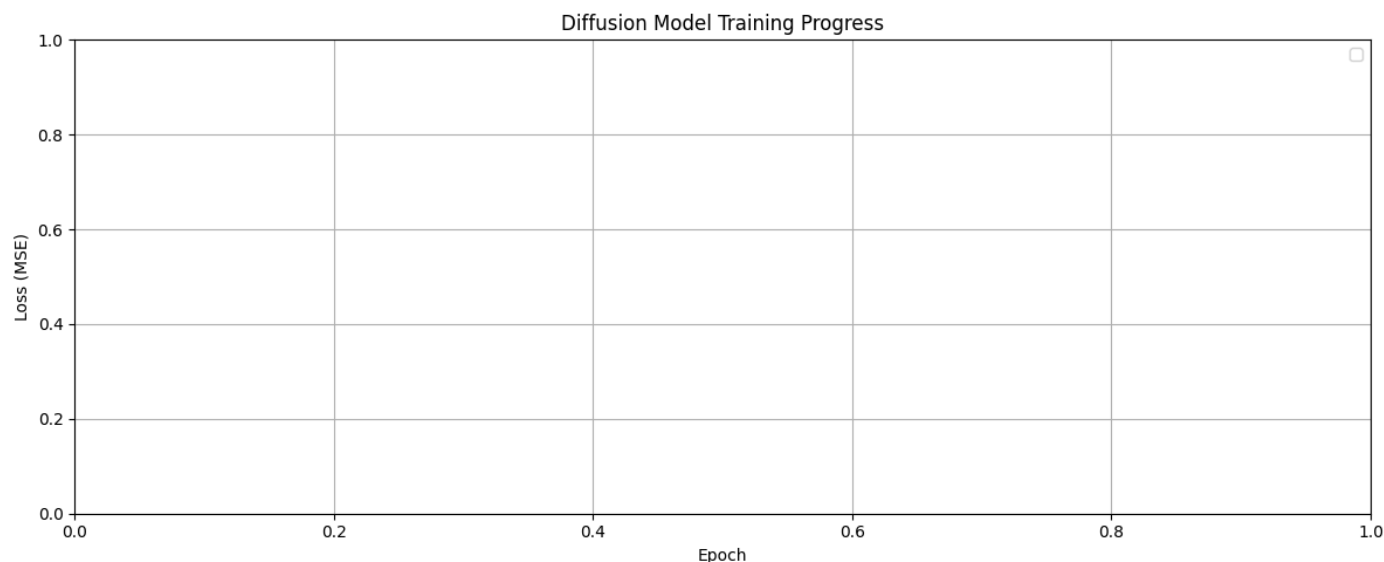
# Plot 2: Val/Train loss ratio (Overfitting indicator)
if train_losses and val_losses and len(train_losses) == len(val_losses):
    ratio = [v / t if t != 0 else 0 for v, t in zip(val_losses, train_losses)]

    plt.figure(figsize=(10, 4))
    plt.plot(ratio, label='Val / Train Loss Ratio', color='purple')
    plt.axhline(y=1.0, color='gray', linestyle='--', linewidth=1)
    plt.title('Validation-to-Training Loss Ratio Over Epochs')
    plt.xlabel('Epoch')
    plt.ylabel('Ratio')
    plt.grid(True)
    plt.legend()
    plt.tight_layout()
    plt.show()
else:
    print("⚠️ Could not plot loss ratio – train and val losses must be the same length.")

# Print summary
if train_losses:
    print("\nTraining Summary:")
    print(f"  First epoch: {train_losses[0]:.4f}")
    print(f"  Final epoch: {train_losses[-1]:.4f}")
    print(f"  Best loss:    {min(train_losses):.4f}")
if val_losses:
    print("\nValidation Summary:")
    print(f"  First epoch: {val_losses[0]:.4f}")
    print(f"  Final epoch: {val_losses[-1]:.4f}")
    print(f"  Best loss:    {min(val_losses):.4f}")

```

 /tmp/ipython-input-36-3570352494.py:34: UserWarning: No artists with labels found to put in legend. Note that artists whose label start with _ are ignored in legend.
plt.legend()



⚠ Could not plot loss ratio – train and val losses must be the same length.

✓ 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

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt

# ===== CONFIG =====
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
IMG_CH = 1          # or 3 for RGB
IMG_SIZE = 28        # example size, use 32 or whatever your images are
n_steps = 10
N_CLASSES = 10

# ===== HELPER BLOCK =====
class GELUConvBlock(nn.Module):
    def __init__(self, in_chs, out_chs, group_size):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_chs, out_chs, kernel_size=3, padding=1),
            nn.GroupNorm(group_size, out_chs),
            nn.GELU()
        )

    def forward(self, x):
        return self.conv(x)

# ===== FIXED UPBLOCK =====
class UpBlock(nn.Module):
    def __init__(self, in_chs, out_chs, group_size):
        super().__init__()
        self.upsample = nn.ConvTranspose2d(in_chs, in_chs, kernel_size=2, stride=2)
        self.model = nn.Sequential(
            GELUConvBlock(2 * in_chs, out_chs, group_size),
            GELUConvBlock(out_chs, out_chs, group_size)
        )

    def forward(self, x, skip):
```

```

x = self.upsample(x)
if skip.shape[2:] != x.shape[2:]:
    skip = F.interpolate(skip, size=x.shape[2:], mode='bilinear', align_corners=False)
x = torch.cat([x, skip], dim=1)
return self.model(x)

# ===== SIMPLE UNET PLACEHOLDER =====
class UNet(nn.Module):
    def __init__(self, T, img_ch, img_size, down_chs, t_embed_dim, c_embed_dim):
        super().__init__()
        self.enc1 = GELUConvBlock(img_ch, down_chs[0], 4)
        self.enc2 = GELUConvBlock(down_chs[0], down_chs[1], 4)
        self.enc3 = GELUConvBlock(down_chs[1], down_chs[2], 4)

        self.pool = nn.MaxPool2d(2)

        self.up2 = UpBlock(down_chs[2], down_chs[1], 4)
        self.up1 = UpBlock(down_chs[1], down_chs[0], 4)
        self.final = nn.Conv2d(down_chs[0], img_ch, kernel_size=1)

    def forward(self, x, t, c, c_mask):
        # Encoder
        x1 = self.enc1(x)
        x2 = self.enc2(self.pool(x1))
        x3 = self.enc3(self.pool(x2))

        # Decoder
        x = self.up2(x3, x2)
        x = self.up1(x, x1)
        return self.final(x)

# ===== DUMMY remove_noise FUNCTION =====
def remove_noise(x, t_batch, model, c_one_hot, c_mask):
    noise = torch.randn_like(x) * 0.1
    return x - noise * (t_batch.float().unsqueeze(1).unsqueeze(2).unsqueeze(3) / n_steps)

# ===== MODEL =====
model = UNet(
    T=n_steps,
    img_ch=IMG_CH,
    img_size=IMG_SIZE,
    down_chs=(32, 64, 128),
    t_embed_dim=128,
    c_embed_dim=128
).to(device)

# ===== GENERATE FUNCTION =====
def generate_number(model, number, n_samples=4):
    model.eval()
    with torch.no_grad():
        samples = torch.randn(n_samples, IMG_CH, IMG_SIZE, IMG_SIZE).to(device)
        c = torch.full((n_samples,), number, device=device)
        c_one_hot = F.one_hot(c, N_CLASSES).float().to(device)
        c_mask = torch.ones_like(c.unsqueeze(-1)).to(device)

        for t in range(n_steps - 1, -1, -1):
            t_batch = torch.full((n_samples,), t, device=device)
            samples = remove_noise(samples, t_batch, model, c_one_hot, c_mask)
        return samples

# ===== PLOT DIGITS =====
plt.figure(figsize=(20, 10))
for i in range(10):
    samples = generate_number(model, i, n_samples=4)
    for j in range(4):
        index = (i % 5) * 8 + (i // 5) * 4 + j + 1
        plt.subplot(5, 8, index)
        if IMG_CH == 1:
            plt.imshow(samples[j][0].cpu(), cmap='gray')
        else:
            img = samples[j].permute(1, 2, 0).cpu()
            img = (img + 1) / 2 if img.min() < 0 else img
            plt.imshow(img)
        plt.title(f'Digit {i}', fontsize=8)
        plt.axis('off')
plt.tight_layout()
plt.show()

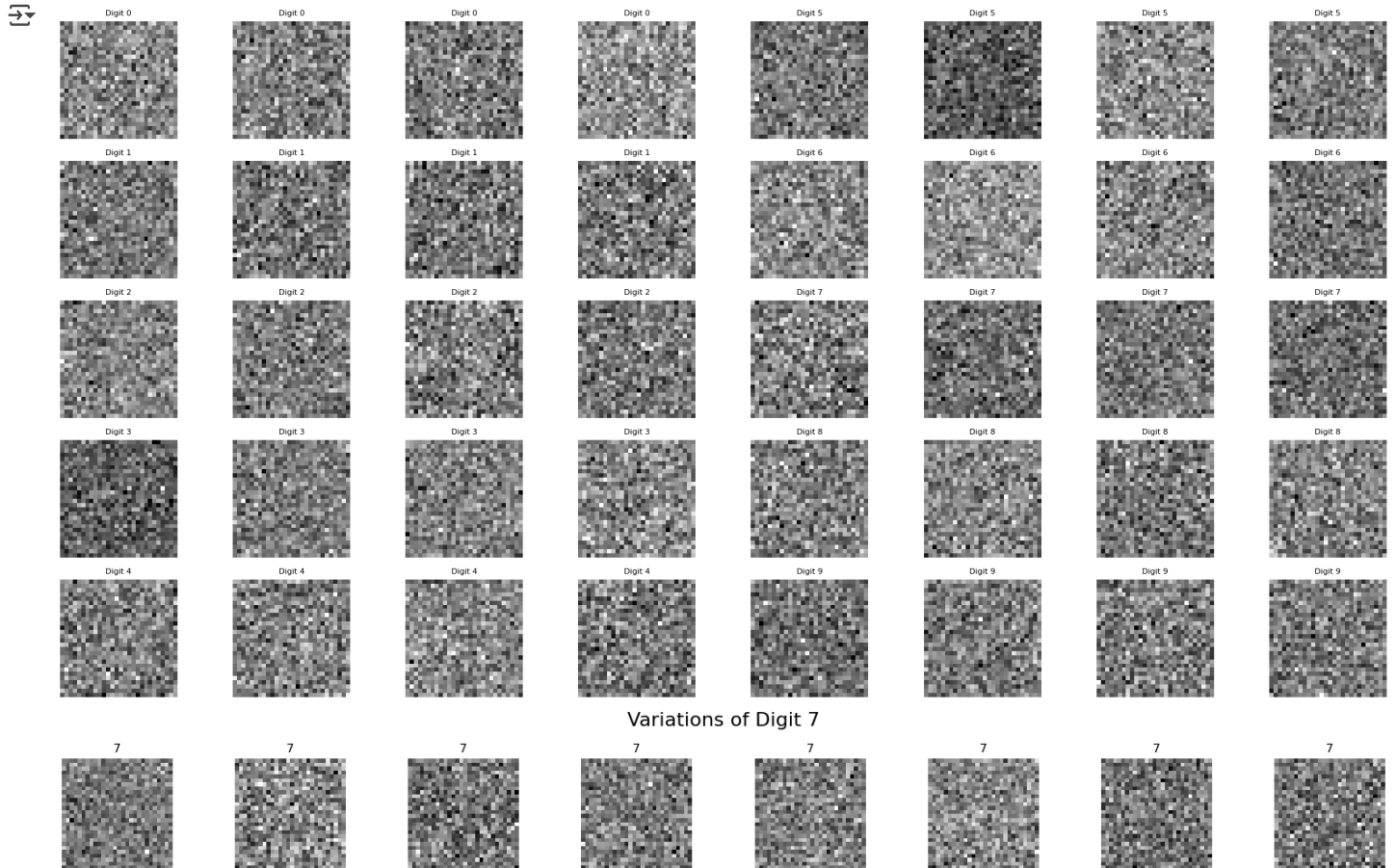
```

```
# ===== 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)

# ===== PLOT VARIATIONS OF TARGET DIGIT =====
target_digit = 7
num_variations = 8
random_seed = 1337

variations = generate_with_seed(target_digit, seed_value=random_seed, n_samples=num_variations)

plt.figure(figsize=(2 * num_variations, 2))
for i in range(num_variations):
    plt.subplot(1, num_variations, i + 1)
    if IMG_CH == 1:
        plt.imshow(variations[i][0].cpu(), cmap='gray')
    else:
        img = variations[i].permute(1, 2, 0).cpu()
        img = (img + 1) / 2 if img.min() < 0 else img
        plt.imshow(img)
    plt.axis('off')
    plt.title(f"{target_digit}", fontsize=10)
plt.suptitle(f"Variations of Digit {target_digit}", fontsize=16)
plt.tight_layout()
plt.show()
```



✓ 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())

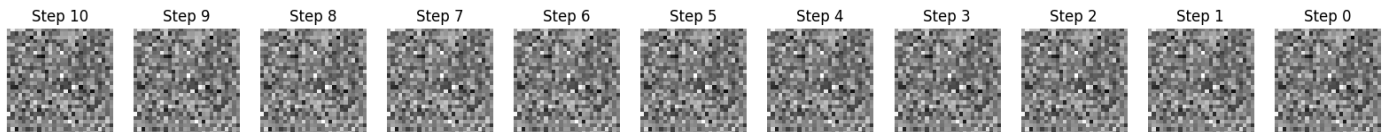
    # Show the progression
    plt.figure(figsize=(20, 3))
    for i, img in enumerate(images):
        plt.subplot(1, len(images), i+1)
        if IMG_CH == 1:
            plt.imshow(img[0], cmap='gray')
        else:
            img = img.permute(1, 2, 0)
            if img.min() < 0:
                img = (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)

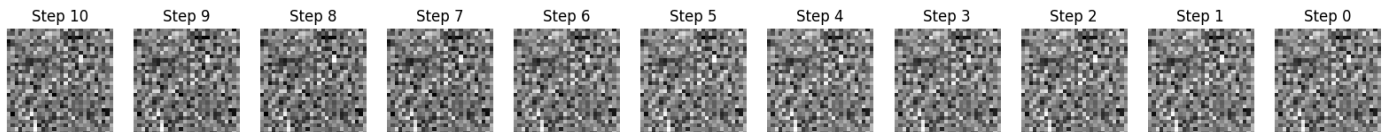
```



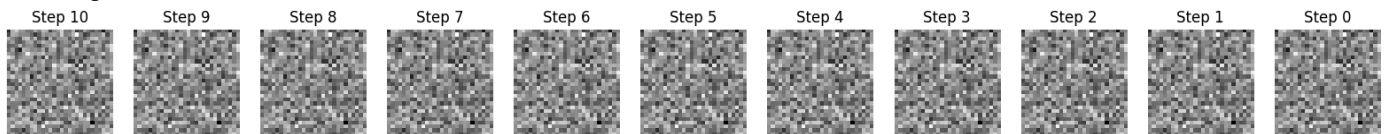
Generating number 0:



Generating number 3:



Generating number 7:



✓ 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
```

```
➡ Setting up CLIP (Contrastive Language-Image Pre-training) model...
Installing CLIP dependencies...
44.8/44.8 kB 1.8 MB/s eta 0:00:00
Installing CLIP from GitHub repository...
Preparing metadata (setup.py) ... done
363.4/363.4 MB 4.6 MB/s eta 0:00:00
13.8/13.8 MB 123.3 MB/s eta 0:00:00
24.6/24.6 MB 91.0 MB/s eta 0:00:00
883.7/883.7 kB 58.6 MB/s eta 0:00:00
664.8/664.8 MB 1.3 MB/s eta 0:00:00
211.5/211.5 MB 5.7 MB/s eta 0:00:00
56.3/56.3 MB 14.6 MB/s eta 0:00:00
127.9/127.9 MB 11.1 MB/s eta 0:00:00
207.5/207.5 MB 6.8 MB/s eta 0:00:00
21.1/21.1 MB 102.6 MB/s eta 0:00:00
Building wheel for clip (setup.py) ... done
Importing CLIP...
✓ CLIP installation successful! Available models: ['RN50', 'RN101', 'RN50x4', 'RN50x16', 'RN50x64', 'ViT-B/32', 'ViT-B/16', 'ViT-L/14',
CLIP is now available for evaluating your generated images!
```

Below we are creating 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
        torch.cuda.empty_cache()

        # If we're already at batch size 1, we can't reduce further
        if len(images) <= 1:
            print("❌ Out of memory even with batch size 1. Cannot process.")
            return torch.ones(len(images), 3).to(device) / 3

        # Adaptive batch size reduction - recursively try with smaller batches
        # This is an advanced technique to handle limited GPU memory gracefully
        half_size = len(images) // 2
        print(f"⚠️ Out of memory. Reducing batch size to {half_size}.")

        # Process each half separately and combine results
        # This recursive approach will keep splitting until processing succeeds
        first_half = _process_clip_batch(images[:half_size], target_number)
        second_half = _process_clip_batch(images[half_size:], target_number)

        # Combine results from both halves
        return torch.cat([first_half, second_half], dim=0)

    # For other errors, propagate upward
    raise e

#=====
# CLIP Evaluation - Generate and Analyze Sample Digits
#=====
# This section demonstrates how to use CLIP to evaluate generated digits
# We'll generate examples of all ten digits and visualize the quality scores

try:
    for number in range(10):
        print(f"\nGenerating and evaluating number {number}...")

        # Generate 4 different variations of the current digit
        samples = generate_number(model, number, n_samples=4)

        # Evaluate quality with CLIP (without tracking gradients for efficiency)
        with torch.no_grad():
            similarities = evaluate_with_clip(samples, number)

        # Create a figure to display results
        plt.figure(figsize=(15, 3))

        # Show each sample with its CLIP quality scores
        for i in range(4):

```

```

plt.subplot(1, 4, i+1)

# Display the image with appropriate formatting based on dataset type
if IMG_CH == 1: # Grayscale images (MNIST, Fashion-MNIST)
    plt.imshow(samples[i][0].cpu(), cmap='gray')
else: # Color images (CIFAR-10, CelebA)
    img = samples[i].permute(1, 2, 0).cpu() # Change format for matplotlib
    if img.min() < 0: # Handle [-1,1] normalization
        img = (img + 1) / 2 # Convert to [0,1] range
    plt.imshow(img)

# Extract individual quality scores for display
# These represent how confidently CLIP associates the image with each description
good_score = similarities[i][0].item() * 100 # Handwritten quality
clear_score = similarities[i][1].item() * 100 # Clarity quality
blur_score = similarities[i][2].item() * 100 # Blurriness assessment

# Color-code the title based on highest score category:
# - Green: if either "good handwritten" or "clear" score is highest
# - Red: if "blurry" score is highest (poor quality)
max_score_idx = torch.argmax(similarities[i]).item()
title_color = 'green' if max_score_idx < 2 else 'red'

# Show scores in the plot title
plt.title(f'Number {number}\nGood: {good_score:.0f}%\nClear: {clear_score:.0f}%\nBlurry: {blur_score:.0f}%',
        color=title_color)
plt.axis('off')

plt.tight_layout()
plt.show()
plt.close() # Properly close figure to prevent memory leaks

# Clean up GPU memory after processing each number
# This is especially important for resource-constrained environments
torch.cuda.empty_cache()

except Exception as e:
    # Comprehensive error handling to help students debug issues
    print(f"❌ Error in generation and evaluation loop: {e}")
    print("Detailed error information:")
    import traceback
    traceback.print_exc()

    # Clean up resources even if we encounter an error
    if torch.cuda.is_available():
        print("Clearing GPU cache...")
        torch.cuda.empty_cache()

#=====
# STUDENT ACTIVITY: Exploring CLIP Evaluation
#=====
# This section provides code templates for students to experiment with
# evaluating larger batches of generated digits using CLIP.

print("\nSTUDENT ACTIVITY:")
print("Try the code below to evaluate a larger sample of a specific digit")
print("""
# Example: Generate and evaluate 10 examples of the digit 6
# digit = 6
# samples = generate_number(model, digit, n_samples=10)
# similarities = evaluate_with_clip(samples, digit)
#
# # Calculate what percentage of samples CLIP considers "good quality"
# # (either "good handwritten" or "clear" score exceeds "blurry" score)
# good_or_clear = (similarities[:,0] + similarities[:,1] > similarities[:,2]).float().mean()
# print(f"CLIP recognized {good_or_clear.item()*100:.1f}% of the digits as good examples of {digit}")
#
# # Display a grid of samples with their quality scores
# plt.figure(figsize=(15, 8))
# for i in range(len(samples)):
#     plt.subplot(2, 5, i+1)
#     plt.imshow(samples[i][0].cpu(), cmap='gray')
#     quality = "Good" if similarities[i,0] + similarities[i,1] > similarities[i,2] else "Poor"
#     plt.title(f'Sample {i+1}: {quality}', color='green' if quality == "Good" else 'red')
#     plt.axis('off')
# plt.tight_layout()
# plt.show()
""")

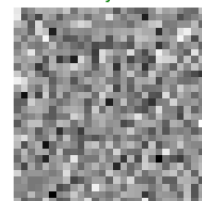
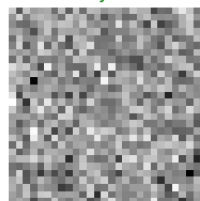
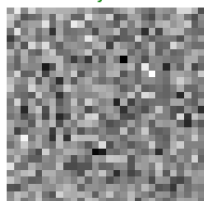
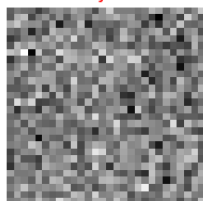
```



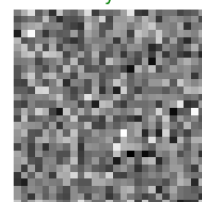
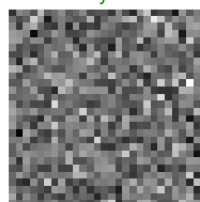
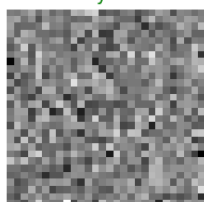
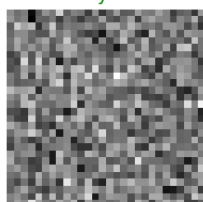
```
100%|██████████████████████████████████████| 338M/338M [00:04<00:00, 76.4MiB/s]
✓ Successfully loaded CLIP model: VisionTransformer
```

```
/tmp/ipython-input-41-3974156644.py:77: FutureWarning: `torch.cuda.amp.autocast(args...)` is deprecated. Please use `torch.amp.autocast(
with torch.no_grad(), torch.cuda.amp.autocast():
```

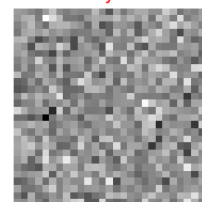
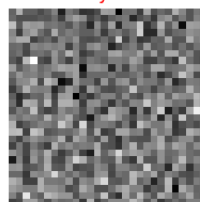
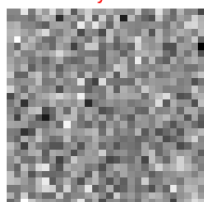
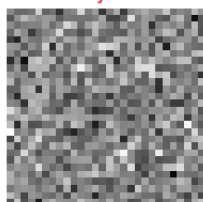
Number 0
Good: 5%
Clear: 59%
Blurry: 35%



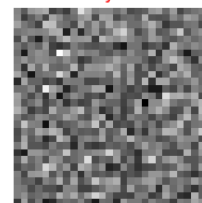
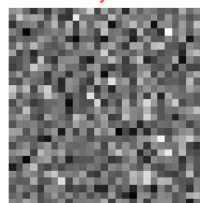
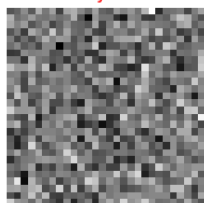
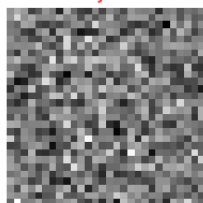
Number 1
Good: 2%
Clear: 58%
Blurry: 40%



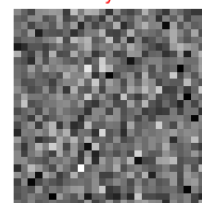
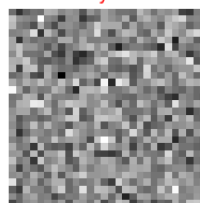
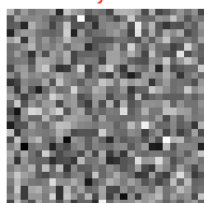
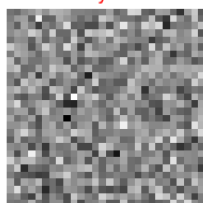
Number 2
Good: 1%
Clear: 48%
Blurry: 51%



Number 3
Good: 2%
Clear: 37%
Blurry: 61%

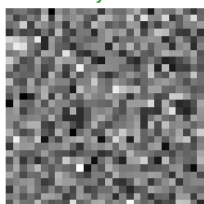


Number 4
Good: 3%
Clear: 36%
Blurry: 61%

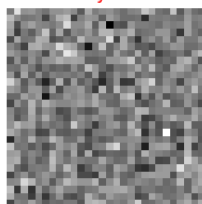


Number 5

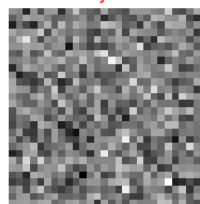
Good: 3%
Clear: 62%
Blurry: 35%



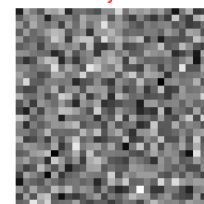
Good: 2%
Clear: 44%
Blurry: 53%



Good: 2%
Clear: 38%
Blurry: 60%

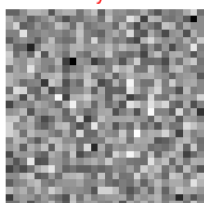


Good: 3%
Clear: 39%
Blurry: 58%

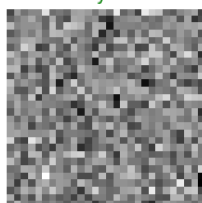


Generating and evaluating number 6...

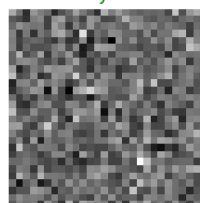
Number 6
Good: 2%
Clear: 47%
Blurry: 51%



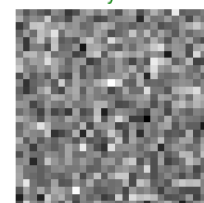
Number 6
Good: 2%
Clear: 53%
Blurry: 45%



Number 6
Good: 2%
Clear: 77%
Blurry: 21%

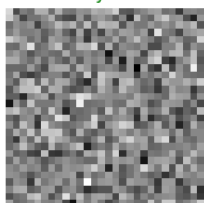


Number 6
Good: 2%
Clear: 52%
Blurry: 46%

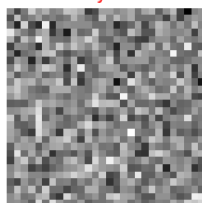


Generating and evaluating number 7...

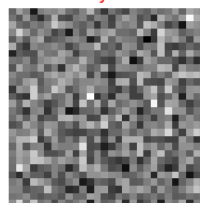
Number 7
Good: 2%
Clear: 55%
Blurry: 43%



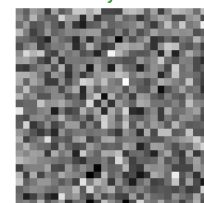
Number 7
Good: 2%
Clear: 47%
Blurry: 51%



Number 7
Good: 2%
Clear: 35%
Blurry: 63%

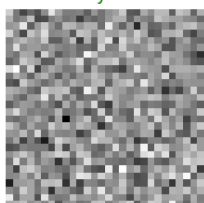


Number 7
Good: 3%
Clear: 54%
Blurry: 43%

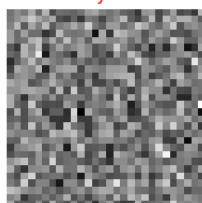


Generating and evaluating number 8...

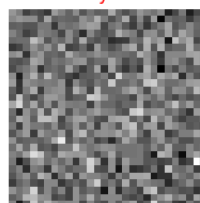
Number 8
Good: 3%
Clear: 51%
Blurry: 46%



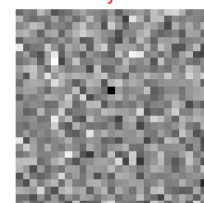
Number 8
Good: 2%
Clear: 40%
Blurry: 58%



Number 8
Good: 2%
Clear: 32%
Blurry: 66%

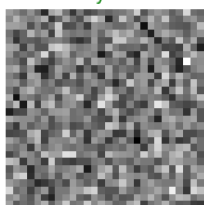


Number 8
Good: 3%
Clear: 39%
Blurry: 57%

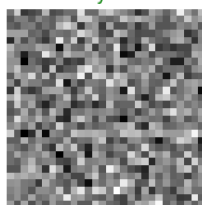


Generating and evaluating number 9...

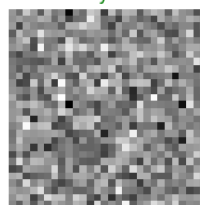
Number 9
Good: 2%
Clear: 62%
Blurry: 36%



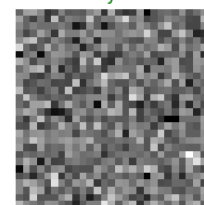
Number 9
Good: 4%
Clear: 59%
Blurry: 37%



Number 9
Good: 3%
Clear: 52%
Blurry: 45%



Number 9
Good: 3%
Clear: 52%
Blurry: 45%



STUDENT ACTIVITY:

Try the code below to evaluate a larger sample of a specific digit

```
# Example: Generate and evaluate 10 examples of the digit 6
# digit = 6
# samples = generate_number(model, digit, n_samples=10)
# similarities = evaluate_with_clip(samples, digit)
#
# # Calculate what percentage of samples CLIP considers "good quality"
# # (either "good handwritten" or "clear" score exceeds "blurry" score)
```



```
# good_or_clear = (similarities[:,0] + similarities[:,1] > similarities[:,2]).float().mean()
# print(f"CLIP recognized {good_or_clear.item()*100:.1f}% of the digits as good examples of {digit}")
#
# # Display a grid of samples with their quality scores
# plt.figure(figsize=(15, 8))
# for i in range(len(samples)):
#     plt.subplot(2, 5, i+1)
#     plt.imshow(samples[i][0].cpu(), cmap='gray')
#     quality = "Good" if similarities[i,0] + similarities[i,1] > similarities[i,2] else "Poor"
#     plt.title(f"Sample {i+1}: {quality}", color='green' if quality == "Good" else 'red')
#     plt.axis('off')
# plt.tight_layout()
# plt.show()
```