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.

```python
# 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:


#--------------------------------------------
# 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:

"""

{"type":"string"}

#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


# 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
```

```python
# Enter your code here:

import torch

# Example dataset selection (make sure this is uncommented if using MNIST)
dataset = 'MNIST'  # Change this if using a different dataset

# 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
    """)

# Minimum GPU memory (in bytes) required for each dataset
gpu_memory_requirements = {
    'MNIST': 2 * 1024**3,        # 2 GB
    'Fashion-MNIST': 2 * 1024**3,
    'CIFAR-10': 4 * 1024**3,
    'CelebA': 8 * 1024**3
}

# Check GPU availability and memory
if torch.cuda.is_available():
    total_memory = torch.cuda.get_device_properties(0).total_memory
    required_memory = gpu_memory_requirements.get(dataset)

    if required_memory is None:
        raise ValueError(f"⚠ Unknown dataset: {dataset}")

    if total_memory < required_memory:
        raise MemoryError(f"""
        □ ERROR: Not enough GPU memory for {dataset}!
        Required: {required_memory / 1024**3:.1f} GB
        Available: {total_memory / 1024**3:.1f} GB
        """)
    else:
        print(f"□ GPU is available with {total_memory / 1024**3:.1f} GB memory. Proceeding with {dataset}.")
else:
    print("⚠ WARNING: No GPU available. Training may be slow on CPU.")
```

```
□ GPU is available with 14.7 GB memory. Proceeding with MNIST.
```

```
#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:


#======================================================================
==========
# 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:


# 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:

import torch
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, random_split

# Constants
BATCH_SIZE = 64  # You can change this as needed
SEED = 42

# Set transform for MNIST
transform = transforms.ToTensor()

# Download and load the dataset
dataset = datasets.MNIST(root='./data', train=True, download=True,
transform=transform)

# Check a sample batch
```

```python
sample_loader = DataLoader(dataset, batch_size=1)
sample_batch = next(iter(sample_loader))
images, labels = sample_batch

print("□ Sample Batch Information:")
print(f"Image shape: {images.shape}")        # Expected: torch.Size([1, 1, 28, 28])
print(f"Label: {labels.item()}")             # Expected: 0-9
print(f"Image dtype: {images.dtype}")        # Expected: torch.float32
print(f"Image pixel range: [{images.min().item()}, {images.max().item()}]")  # Expected: [0.0, 1.0]

# Fix random seed for reproducibility
generator = torch.Generator().manual_seed(SEED)

# Calculate split sizes
train_size = int(0.8 * len(dataset))
val_size = len(dataset) - train_size

# Perform the split
train_dataset, val_dataset = random_split(dataset, [train_size, val_size], generator=generator)

print(f"□ Train size: {len(train_dataset)}, Validation size: {len(val_dataset)}")

import os

# Use the number of available CPU cores for data loading
num_workers = os.cpu_count()

# Create DataLoaders
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=num_workers)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=num_workers)

print("□ DataLoaders created.")
```

```
□ Sample Batch Information:
Image shape: torch.Size([1, 1, 28, 28])
Label: 5
Image dtype: torch.float32
Image pixel range: [0.0, 1.0]
□ Train size: 48000, Validation size: 12000
□ 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

```
import torch
import torch.nn as 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 and group_size > 1:
                group_size -= 1
            print(f"□ Adjusted group_size to {group_size}")

        # Define the convolutional block
        self.block = nn.Sequential(
            nn.Conv2d(in_channels=in_ch, out_channels=out_ch,
kernel_size=3, padding=1, bias=False),
            nn.GroupNorm(num_groups=group_size, num_channels=out_ch),
            nn.GELU()
        )

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

import torch
import torch.nn as nn
```

```python
from einops.layers.torch import Rearrange

# Assume GELUConvBlock is already defined and imported
# from previous code block

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__()

        # Rearranging from (B, C, H, W) to (B, 4*C, H/2, W/2)
        self.rearrange = Rearrange('b c (h 2) (w 2) -> b (c 4) h w')

        # 4 times the channels due to rearrangement
        self.conv = GELUConvBlock(in_ch=in_chs * 4, out_ch=in_chs,
group_size=group_size)

    def forward(self, x):
        x = self.rearrange(x)  # Downsamples spatial dims, increases
channels
        x = self.conv(x)       # Process through 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
        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)

import torch
import torch.nn as nn

# Make sure GELUConvBlock is already defined and imported

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__()
```

```
        # Transposed convolution to upsample by 2x
        self.upsample = nn.ConvTranspose2d(in_chs, in_chs,
kernel_size=2, stride=2)

        # After concatenation, the channel count doubles
        self.conv_block = nn.Sequential(
            GELUConvBlock(in_ch=in_chs * 2, out_ch=out_chs,
group_size=group_size),
            GELUConvBlock(in_ch=out_chs, out_ch=out_chs,
group_size=group_size)
        )

        print(f"□ Created UpBlock: in_chs={in_chs}, out_chs={out_chs},
spatial_increase=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]
        """
        x = self.upsample(x)  # Upsample to match spatial dims of skip
        # Concatenate along the channel dimension
        x = torch.cat([x, skip], dim=1)
        x = self.conv_block(x)  # Process through GELUConvBlocks
        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
    """
```

```python
    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


import torch
import torch.nn as nn

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

        # Define the embedding transformation: Linear -> GELU ->
Linear -> Unflatten
        self.model = nn.Sequential(
            nn.Linear(input_dim, emb_dim),
```

```
            nn.GELU(),
            nn.Linear(emb_dim, emb_dim),
            nn.Unflatten(dim=1, unflattened_size=(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

# Placeholder for SinusoidalPositionEmbedBlock if not implemented yet
class SinusoidalPositionEmbedBlock(nn.Module):
    def __init__(self, T, emb_dim):
        super().__init__()
        self.T = T
        self.emb_dim = emb_dim

        # Precompute sinusoidal embeddings
        inv_freq = 1.0 / (10000 ** (torch.arange(0, emb_dim,
2).float() / emb_dim))
        self.register_buffer("inv_freq", inv_freq)

    def forward(self, t):
        # t: [B]
        sinusoid_inp = t[:, None].float() * self.inv_freq[None]
        emb = torch.cat([torch.sin(sinusoid_inp),
torch.cos(sinusoid_inp)], dim=-1)
        return emb  # [B, emb_dim]

class UNet(nn.Module):
    def __init__(self, T, img_ch, img_size, down_chs, t_embed_dim,
c_embed_dim):
        super().__init__()

        # Time embedding
```

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

        # Class embedding
        self.class_embed = EmbedBlock(input_dim=c_embed_dim,
emb_dim=t_embed_dim)

        # Initial convolution
        self.input_conv = GELUConvBlock(img_ch, down_chs[0],
group_size=4)

        # Downsampling blocks
        self.down_blocks = nn.ModuleList()
        in_channels = down_chs[0]
        for out_channels in down_chs[1:]:
            block = nn.Sequential(
                RearrangePoolBlock(in_channels, group_size=4),
                GELUConvBlock(in_channels, out_channels, group_size=4)
            )
            self.down_blocks.append(block)
            in_channels = out_channels

        # Middle blocks
        self.middle_block = nn.Sequential(
            GELUConvBlock(in_channels, in_channels, group_size=4),
            GELUConvBlock(in_channels, in_channels, group_size=4)
        )

        # Upsampling blocks
        self.up_blocks = nn.ModuleList()
        reversed_chs = list(reversed(down_chs))
        for i in range(len(reversed_chs) - 1):
            in_chs = reversed_chs[i]
            out_chs = reversed_chs[i + 1]
            self.up_blocks.append(UpBlock(in_chs, out_chs,
group_size=4))

        # Final convolution
        self.output_conv = 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):
        """
```

```
        x: Input image tensor [B, C, H, W]
        t: Time step tensor [B]
        c: Class one-hot or index tensor [B, c_embed_dim]
        c_mask: Binary tensor indicating whether to use class
conditioning [B, 1]
        """
        t_emb = self.time_embed(t)  # [B, t_embed_dim]
        c_emb = self.class_embed(c)  # [B, t_embed_dim, 1, 1]

        # Apply class mask (conditionally zero out)
        c_emb = c_emb * c_mask.unsqueeze(-1).unsqueeze(-1)  # [B,
t_embed_dim, 1, 1]

        # Initial convolution
        x = self.input_conv(x)

        # Downsampling path with skip connections
        skips = []
        for block in self.down_blocks:
            x = block(x)
            skips.append(x)

        # Middle processing + conditioning
        x = self.middle_block(x)
        # Add time and class embeddings
        x = x + t_emb.unsqueeze(-1).unsqueeze(-1) + c_emb

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

        # Final projection
        out = self.output_conv(x)
        return out
```

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 = 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
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

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

    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)
    """
    # 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)  # [B,
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
    x_t = sqrt_alpha_bar_t * x_0 + sqrt_one_minus_alpha_bar_t * noise

    return x_t, noise

# 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 specified timesteps
    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)
    sqrt_one_minus_alpha_bar_t = sqrt_one_minus_alpha_bar[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

import matplotlib.pyplot as plt
import torch
```

```python
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')

    # Add noise progressively
    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)  # Add noise

        # Display 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.tight_layout()
    plt.show()
```

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!

```
import torch
import torch.nn as nn
import torchvision
import matplotlib.pyplot as plt
from torch.optim import Adam
from torch.utils.data import DataLoader, random_split
from torchvision import transforms
import numpy as np
import torch.nn.functional as F

from einops.layers.torch import Rearrange

# ----------------------------
# Hyperparameters and Setup
# ----------------------------
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
# Assuming these are defined in previous cells or need to be set here
# IMG_CH = 1
# IMG_SIZE = 28
# N_CLASSES = 10
# n_steps = 1000
# BATCH_SIZE = 64 # Assuming this is defined in previous cells

# ----------------------------
# Beta Schedule Setup
# ----------------------------
# Assuming n_steps is defined
beta = torch.linspace(1e-4, 0.02, n_steps).to(device)
alpha = (1.0 - beta).to(device)
alpha_bar = torch.cumprod(alpha, dim=0).to(device)

sqrt_alpha_bar = torch.sqrt(alpha_bar).to(device)
sqrt_one_minus_alpha_bar = torch.sqrt(1 - alpha_bar).to(device)

# ----------------------------
# Helper Classes
# ----------------------------

class GELUConvBlock(nn.Module):
    def __init__(self, in_ch, out_ch, group_size):
        super().__init__()
        # Ensure group_size is compatible with out_ch
        if out_ch % group_size != 0:
            group_size = max(1, next(g for g in range(group_size, 0, -
1) if out_ch % g == 0))
```

```
        self.block = nn.Sequential(
            nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=1,
bias=False),
            nn.GroupNorm(group_size, out_ch),
            nn.GELU()
        )

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

# Removed RearrangePoolBlock and replaced with MaxPool2d for
downsampling

class DownBlock(nn.Module):
    def __init__(self, in_chs, out_chs, group_size):
        super().__init__()
        self.conv1 = GELUConvBlock(in_chs, out_chs, group_size)
        self.conv2 = GELUConvBlock(out_chs, out_chs, group_size)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        skip = x # Save for skip connection before pooling
        x = self.pool(x)
        return x, skip


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)
        # Input channels will be in_chs + skip_connection_channels
(which is also in_chs)
        self.conv_block = nn.Sequential(
            GELUConvBlock(in_ch=in_chs + in_chs, out_ch=out_chs,
group_size=group_size),
            GELUConvBlock(in_ch=out_chs, out_ch=out_chs,
group_size=group_size)
        )

    def forward(self, x, skip):
        x = self.upsample(x)
        # Pad x if its spatial dimensions don't exactly match skip's
        # This can happen with some input sizes and pooling/upsampling
combinations
        diff_h = skip.size(2) - x.size(2)
        diff_w = skip.size(3) - x.size(3)
        if diff_h > 0 or diff_w > 0:
```

```
             x = F.pad(x, [diff_w // 2, diff_w - diff_w // 2,
                           diff_h // 2, diff_h - diff_h // 2])

        x = torch.cat([x, skip], dim=1)
        return self.conv_block(x)

class EmbedBlock(nn.Module):
    def __init__(self, input_dim, emb_dim):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(input_dim, emb_dim),
            nn.GELU(),
            nn.Linear(emb_dim, emb_dim),
            nn.Unflatten(1, (emb_dim, 1, 1))
        )

    def forward(self, x):
        # Assuming x is already one-hot encoded or has the correct
last dimension
        return self.model(x.view(x.size(0), -1))


class SinusoidalPositionEmbedBlock(nn.Module):
    def __init__(self, emb_dim): # Corrected constructor
        super().__init__()
        self.register_buffer('inv_freq', 1.0 / (10000 **
(torch.arange(0, emb_dim, 2).float() / emb_dim)))

    def forward(self, t):
        # t: [B]
        sinusoid_inp = t[:, None].float() *
self.inv_freq[None].to(t.device) # Ensure inv_freq is on the same
device
        return torch.cat([torch.sin(sinusoid_inp),
torch.cos(sinusoid_inp)], dim=-1)


class UNet(nn.Module):
    def __init__(self, T, img_ch, img_size, down_chs, t_embed_dim,
c_embed_dim):
        super().__init__()

        # Time embedding
        self.time_embed = nn.Sequential(
            SinusoidalPositionEmbedBlock(t_embed_dim), # Corrected
constructor
            nn.Linear(t_embed_dim, t_embed_dim),
            nn.GELU(),
            nn.Linear(t_embed_dim, t_embed_dim)
        )
```

```
        # Class embedding
        self.class_embed = EmbedBlock(c_embed_dim, t_embed_dim) #
Corrected constructor

        # Initial convolution
        self.input_conv = GELUConvBlock(img_ch, down_chs[0],
group_size=4)

        # Downsampling blocks
        self.down_blocks = nn.ModuleList()
        in_channels = down_chs[0]
        for out_channels in down_chs[1:]:
            self.down_blocks.append(DownBlock(in_channels,
out_channels, group_size=4))
            in_channels = out_channels

        # Middle blocks
        self.middle_block = nn.Sequential(
            GELUConvBlock(in_channels, in_channels, group_size=4),
            GELUConvBlock(in_channels, in_channels, group_size=4)
        )

        # Upsampling blocks
        self.up_blocks = nn.ModuleList()
        reversed_down_chs = list(reversed(down_chs))
        # Create up blocks
        for i in range(len(reversed_down_chs) - 1):
             # Input channels to UpBlock are previous layer output and
skip connection channels
             # Output channels are the next channel size in the
reversed list
             self.up_blocks.append(UpBlock(reversed_down_chs[i],
reversed_down_chs[i+1], group_size=4))


        # Final convolution
        self.output_conv = 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):
        """
        x: Input image tensor [B, C, H, W]
        t: Time step tensor [B]
        c: Class one-hot or index tensor [B, c_embed_dim] (if one-hot)
or [B] (if index)
        c_mask: Binary tensor indicating whether to use class
```

```
conditioning [B]
        """
        t_emb = self.time_embed(t)  # [B, t_embed_dim]

        # Ensure c is one-hot encoded if it's not already
        if c.ndim == 1:
             c_one_hot = F.one_hot(c,
num_classes=N_CLASSES).float().to(c.device)
        else:
             c_one_hot = c # Assume it's already one-hot

        c_emb = self.class_embed(c_one_hot)  # [B, t_embed_dim, 1, 1]

        # Apply class mask (conditionally zero out) - Removed
problematic multiplication
        # The c_mask is available here if needed for classifier-free
guidance during sampling


        # Initial convolution
        x = self.input_conv(x)

        # Downsampling path with skip connections
        skips = []
        for block in self.down_blocks:
            x, skip = block(x)
            skips.append(skip)

        # Middle processing + conditioning
        x = self.middle_block(x)
        # Add time and class embeddings
        # Add c_emb directly, assuming c_mask is handled externally
for sampling
        x = x + t_emb.unsqueeze(-1).unsqueeze(-1) + c_emb


        # Upsampling path with skip connections
        # Start with the output of the middle block
        x = self.up_blocks[0](x, skips[-1]) # First up block with last
skip
        for i in range(1, len(self.up_blocks)):
            # Subsequent up blocks with remaining skips in reverse
order
            x = self.up_blocks[i](x, skips[-(i+1)])


        # Final projection
        out = self.output_conv(x)
        return out
```

```python
# ----------------------------
# Instantiate the Model
# ----------------------------
# Ensure IMG_CH, IMG_SIZE, N_CLASSES, n_steps are defined before this
model = UNet(
    T=n_steps,
    img_ch=IMG_CH,
    img_size=IMG_SIZE,
    down_chs=(32, 64, 128), # Example channel sizes
    t_embed_dim=128, # Increased embedding dimension for time
    c_embed_dim=N_CLASSES # Use N_CLASSES for class embedding input
dimension
).to(device)

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'}")

# ----------------------------
# Count Parameters and Estimate Memory
# ----------------------------
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if
p.requires_grad)

def estimate_model_memory(model, input_size=(1, IMG_CH, IMG_SIZE,
IMG_SIZE)):
    dummy = torch.zeros(*input_size).to(device)
    try:
        from torchinfo import summary
        print(summary(model, input_size=input_size))
    except ImportError:
        print("Install torchinfo (`pip install torchinfo`) for
detailed summary.")
        print(f"Model has {count_parameters(model)/1e6:.2f}M
parameters")
    except Exception as e:
        print(f"Error during model summary or parameter count: {e}")
        print(f"Model has {count_parameters(model)/1e6:.2f}M
parameters")


estimate_model_memory(model)

# ----------------------------
```

```
# Data Integrity Check
# ----------------------------
def check_data_ranges(loader):
    for batch in loader:
        images, labels = batch
        print(f"Image range: min={images.min().item():.2f},
max={images.max().item():.2f}")
        print(f"Label range: min={labels.min().item()},
max={labels.max().item()}")
        break  # Only one batch check

# ----------------------------
# Load MNIST and check
# ----------------------------
# Assuming transform, dataset, train_size, val_size are defined in
previous cells
# If not, define them here:
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

dataset = torchvision.datasets.MNIST(root='./data', train=True,
download=True, transform=transform)
train_size = int(0.8 * len(dataset))
val_size = len(dataset) - train_size
# Ensure SEED is defined for reproducibility
generator = torch.Generator().manual_seed(SEED)
train_ds, val_ds = random_split(dataset, [train_size, val_size],
generator=generator)

# Ensure BATCH_SIZE and num_workers are defined
# Assuming num_workers = os.cpu_count() is run in a previous cell
train_loader = DataLoader(train_ds, batch_size=BATCH_SIZE,
shuffle=True, num_workers=num_workers)
val_loader = DataLoader(val_ds, batch_size=BATCH_SIZE, shuffle=False,
num_workers=num_workers)


check_data_ranges(train_loader)
check_data_ranges(val_loader)

# ----------------------------
# Optimizer and 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
)

# ------------------------------
# Training Functions
# ------------------------------

# 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
    x_t, noise = add_noise(x, t)

    # 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
    loss = F.mse_loss(predicted_noise, noise)

    return loss

# 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:
            img = grid.permute(1, 2, 0).cpu()
            if img.min() < 0:
                img = (img + 1) / 2
            plt.imshow(img)

        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,
```

```python
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.")

# 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)
```

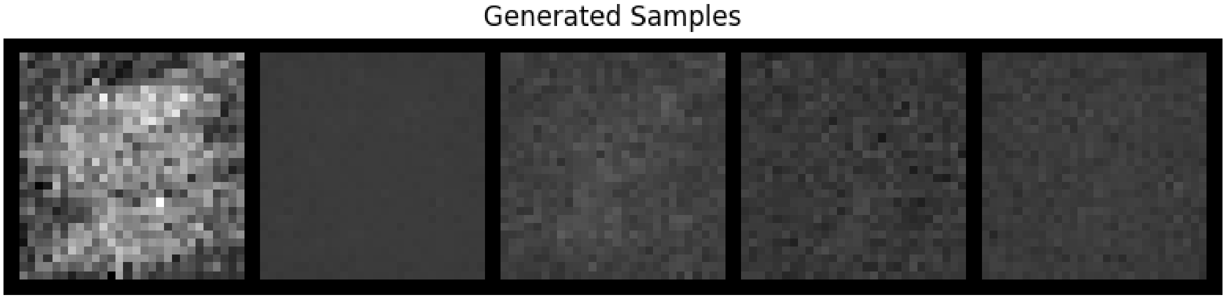
```python
# Progress tracking variables
best_loss = float('inf')
train_losses = []
val_losses = []
no_improve_epochs = 0

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

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_loader):  #
Fixed: dataloader → train_loader
            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 % display_frequency == 0:
                print(f"  Step {step}/{len(train_loader)}, Loss:
{loss.item():.4f}")

                # Generate samples less frequently to save time
                if step % generate_frequency == 0 and step > 0:
                    print("  Generating samples...")
                    generate_samples(model, n_samples=5)
```

```
        # End of epoch - calculate average training loss
        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}")

        # 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_loader:
                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
        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}")

        # 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
        if epoch % 2 == 0 or epoch == EPOCHS - 1:
            print("\nGenerating samples for visual progress check...")
            generate_samples(model, n_samples=10)

        # 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, 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 % 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 Exception as e:
    print(f"An error occurred during training: {e}")
    import traceback
    traceback.print_exc()



# Final wrap-up
print("\n" + "="*50)
print("TRAINING COMPLETE")
print("="*50)
print(f"Best validation loss: {best_loss:.4f}")

# Generate final samples
print("Generating final samples...")
generate_samples(model, n_samples=10)

# Display final loss curves
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()

# Clean up memory
torch.cuda.empty_cache()
```

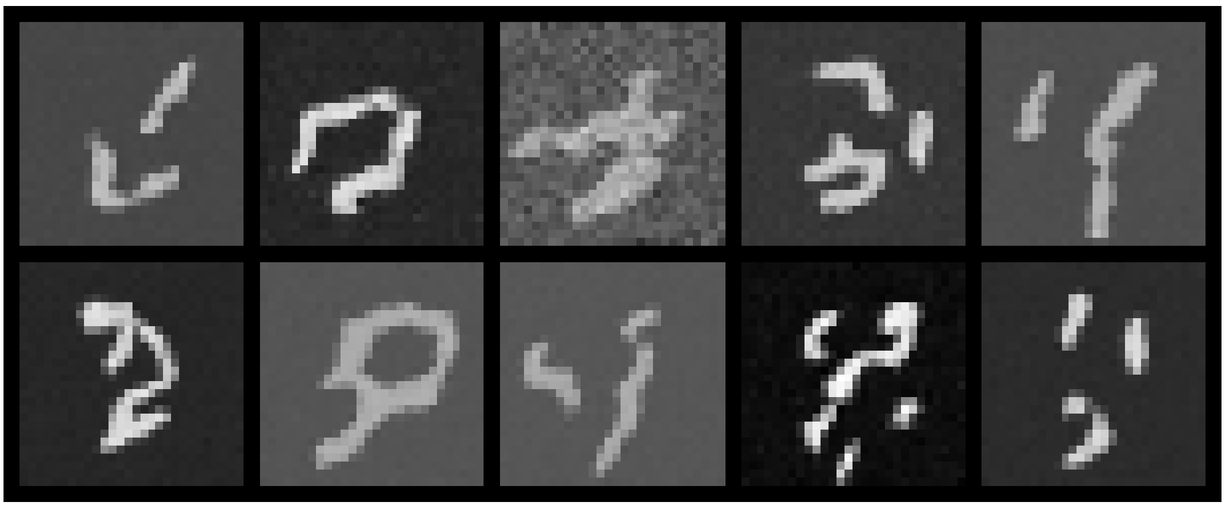
```
□ Created UNet with 3 scale levels
□ Channel dimensions: (32, 64, 128)

==================================================
MODEL ARCHITECTURE SUMMARY
==================================================
Input resolution: 28x28
Input channels: 1
Time steps: 1000
Condition classes: 10
GPU acceleration: Yes
Install torchinfo (`pip install torchinfo`) for detailed summary.
Model has 0.94M parameters
Image range: min=-1.00, max=1.00
Label range: min=0, max=9
Image range: min=-1.00, max=1.00
Label range: min=0, max=9

==================================================
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 get_last_lr() to access the learning rate.
  warnings.warn(

  Step 0/750, Loss: 1.0230
  Step 100/750, Loss: 0.0443
  Step 200/750, Loss: 0.0415
  Step 300/750, Loss: 0.0370
  Step 400/750, Loss: 0.0355
  Step 500/750, Loss: 0.0399
  Generating samples...
```

Generated Samples

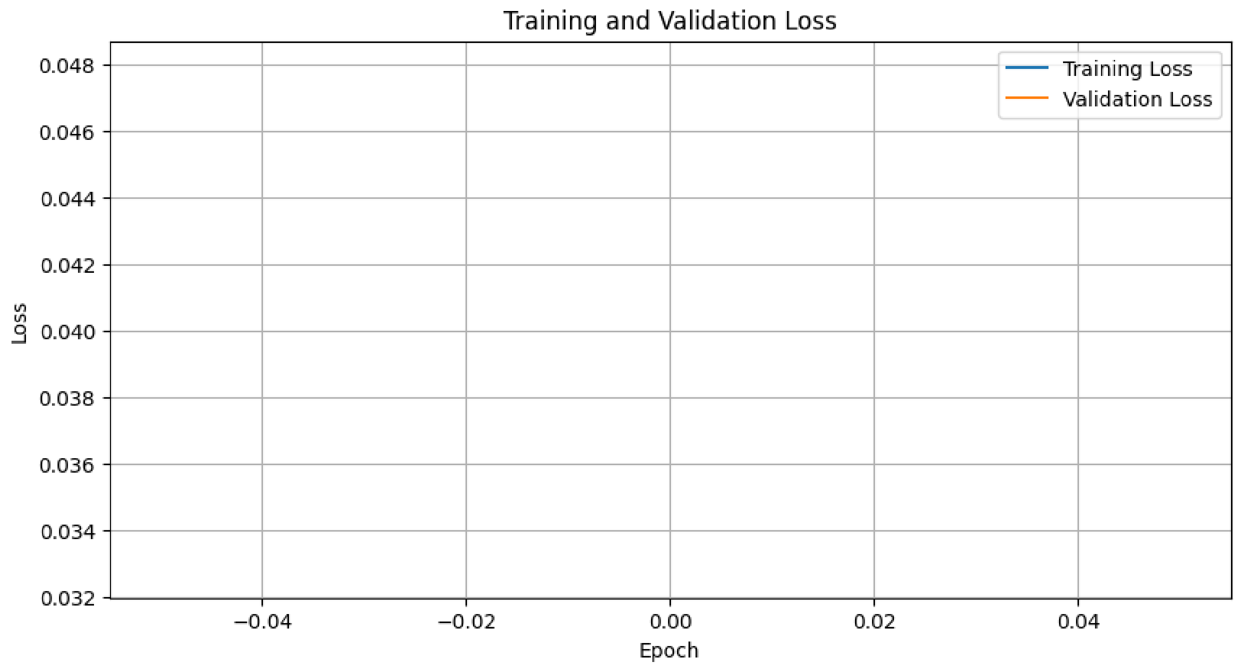


```
  Step 600/750, Loss: 0.0458
  Step 700/750, Loss: 0.0332

Training - Epoch 1 average loss: 0.0479
Running validation...
Validation - Epoch 1 average loss: 0.0327
Learning rate: 0.001000

Generating samples for visual progress check...
```

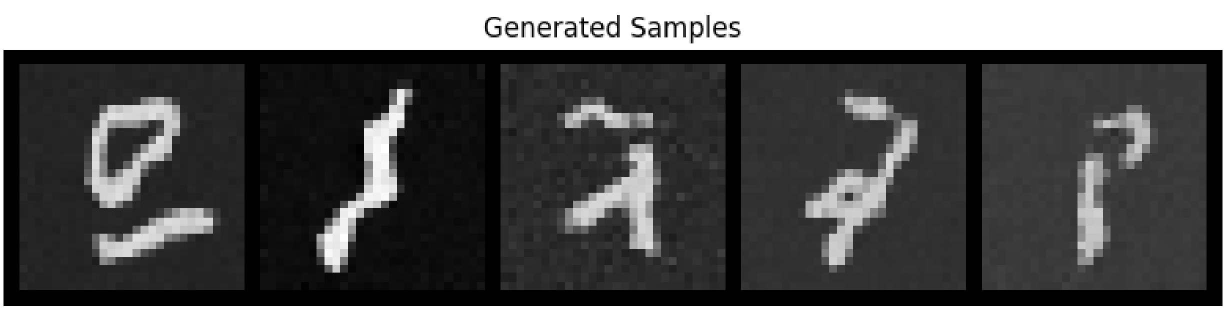
Generated Samples



```
Created backup at best_diffusion_model.pt.backup
Model successfully saved to best_diffusion_model.pt
✓ New best model saved! (Val Loss: 0.0327)
```



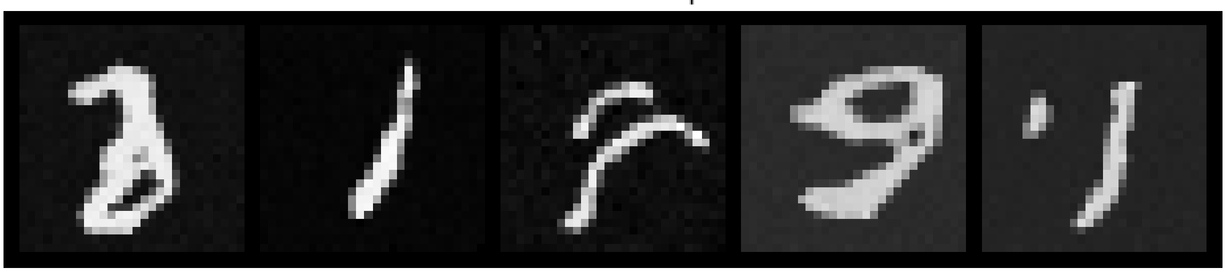

```
Epoch 2/30
--------------------
  Step 0/750, Loss: 0.0409
  Step 100/750, Loss: 0.0339
  Step 200/750, Loss: 0.0369
  Step 300/750, Loss: 0.0341
  Step 400/750, Loss: 0.0311
  Step 500/750, Loss: 0.0353
  Generating samples...
```




```
  Step 600/750, Loss: 0.0338
  Step 700/750, Loss: 0.0283

Training - Epoch 2 average loss: 0.0318
Running validation...
Validation - Epoch 2 average loss: 0.0304
Learning rate: 0.001000
```

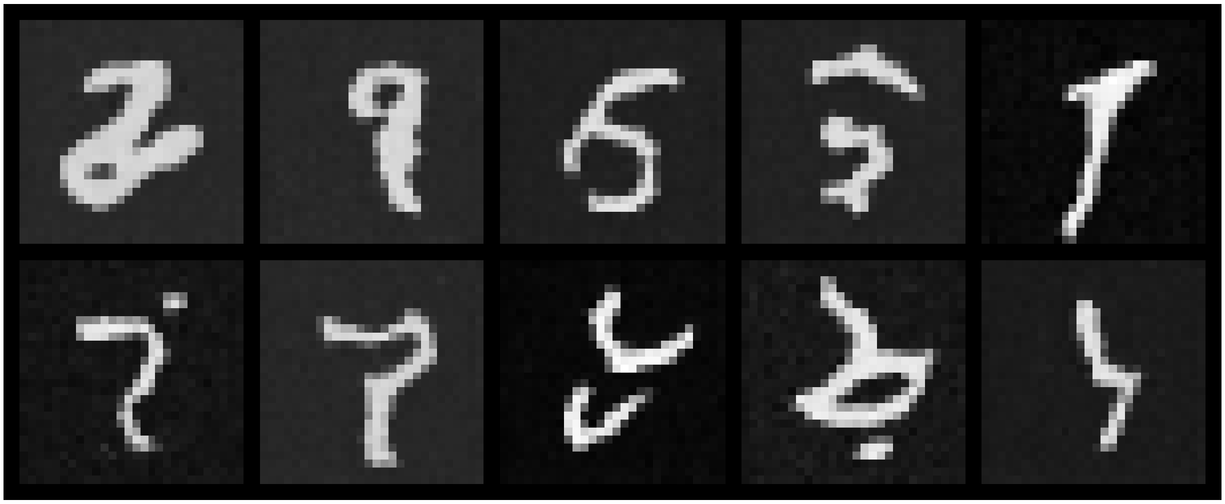
```
Created backup at best_diffusion_model.pt.backup
Model successfully saved to best_diffusion_model.pt
✓ New best model saved! (Val Loss: 0.0304)

Epoch 3/30
--------------------
  Step 0/750, Loss: 0.0232
  Step 100/750, Loss: 0.0331
  Step 200/750, Loss: 0.0361
  Step 300/750, Loss: 0.0297
  Step 400/750, Loss: 0.0405
  Step 500/750, Loss: 0.0251
  Generating samples...
```

Generated Samples



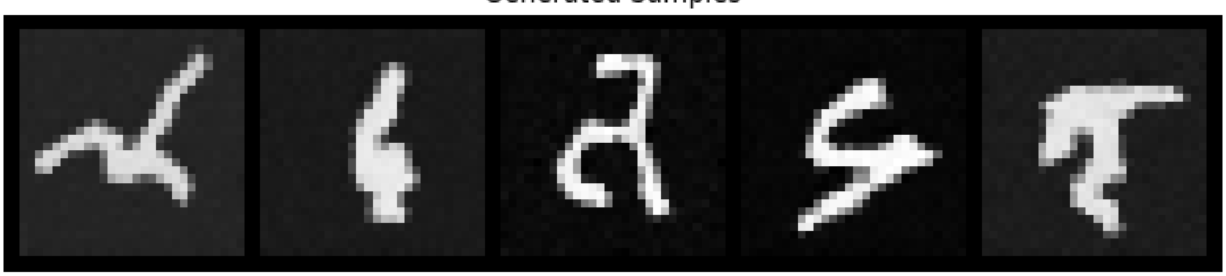
```
  Step 600/750, Loss: 0.0300
  Step 700/750, Loss: 0.0247

Training - Epoch 3 average loss: 0.0299
Running validation...
Validation - Epoch 3 average loss: 0.0309
Learning rate: 0.001000

Generating samples for visual progress check...
```

Generated Samples



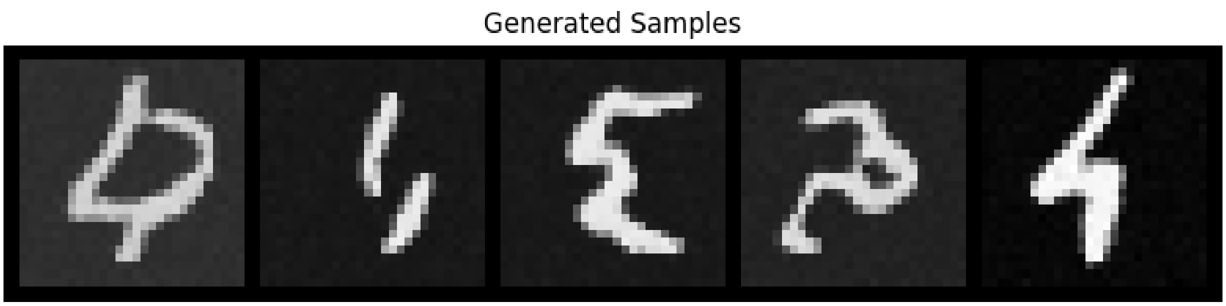
```
No improvement for 1/10 epochs

Epoch 4/30
--------------------
  Step 0/750, Loss: 0.0345
  Step 100/750, Loss: 0.0303
  Step 200/750, Loss: 0.0343
  Step 300/750, Loss: 0.0250
  Step 400/750, Loss: 0.0188
  Step 500/750, Loss: 0.0319
  Generating samples...
```

Generated Samples

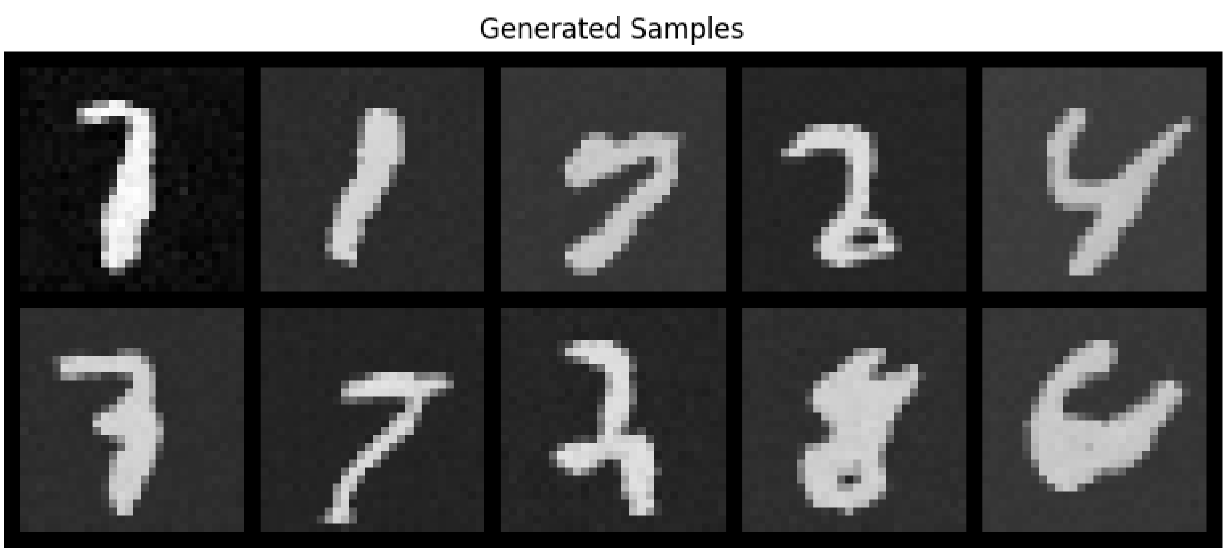


```
  Step 600/750, Loss: 0.0256
  Step 700/750, Loss: 0.0297

Training - Epoch 4 average loss: 0.0282
Running validation...
Validation - Epoch 4 average loss: 0.0281
Learning rate: 0.001000
Created backup at best_diffusion_model.pt.backup
Model successfully saved to best_diffusion_model.pt
✓ New best model saved! (Val Loss: 0.0281)
```

```
Epoch 5/30
--------------------
  Step 0/750, Loss: 0.0289
  Step 100/750, Loss: 0.0304
  Step 200/750, Loss: 0.0257
  Step 300/750, Loss: 0.0280
  Step 400/750, Loss: 0.0261
  Step 500/750, Loss: 0.0326
  Generating samples...
```

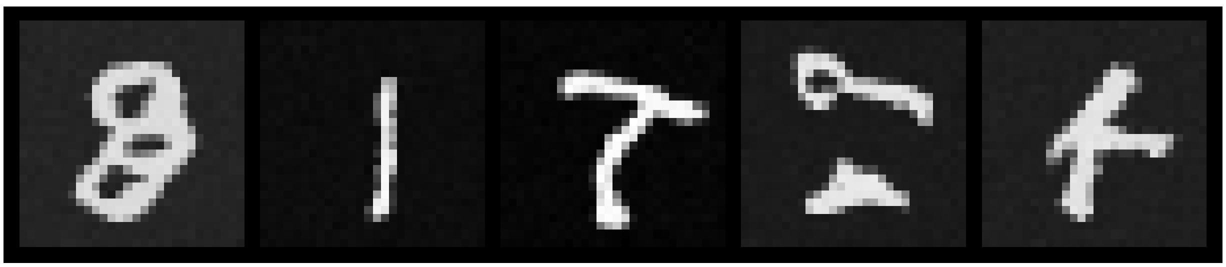



```
  Step 600/750, Loss: 0.0290
  Step 700/750, Loss: 0.0302

Training - Epoch 5 average loss: 0.0276
Running validation...
Validation - Epoch 5 average loss: 0.0279
Learning rate: 0.001000

Generating samples for visual progress check...
```



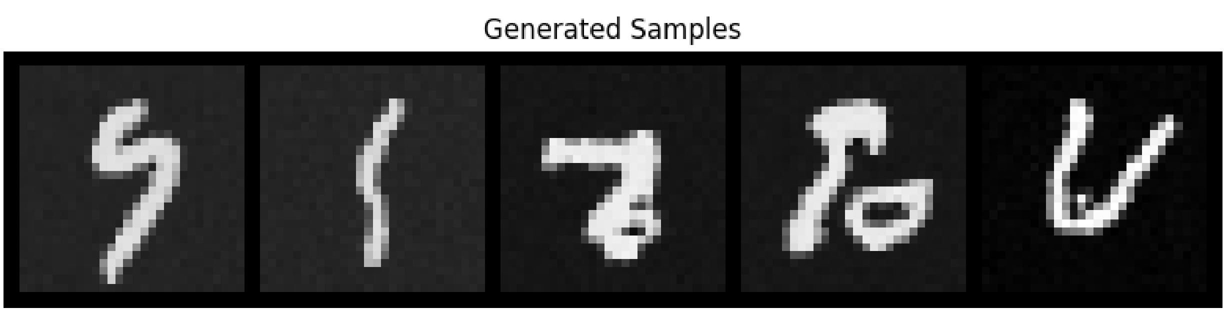
```
Created backup at best_diffusion_model.pt.backup
Model successfully saved to best_diffusion_model.pt
✓ New best model saved! (Val Loss: 0.0279)

Epoch 6/30
--------------------
  Step 0/750, Loss: 0.0267
  Step 100/750, Loss: 0.0193
  Step 200/750, Loss: 0.0276
  Step 300/750, Loss: 0.0272
  Step 400/750, Loss: 0.0240
  Step 500/750, Loss: 0.0218
  Generating samples...
```

Generated Samples



```
  Step 600/750, Loss: 0.0264
  Step 700/750, Loss: 0.0173

Training - Epoch 6 average loss: 0.0270
Running validation...
Validation - Epoch 6 average loss: 0.0268
Learning rate: 0.001000
Created backup at best_diffusion_model.pt.backup
Model successfully saved to best_diffusion_model.pt
✓ New best model saved! (Val Loss: 0.0268)
```

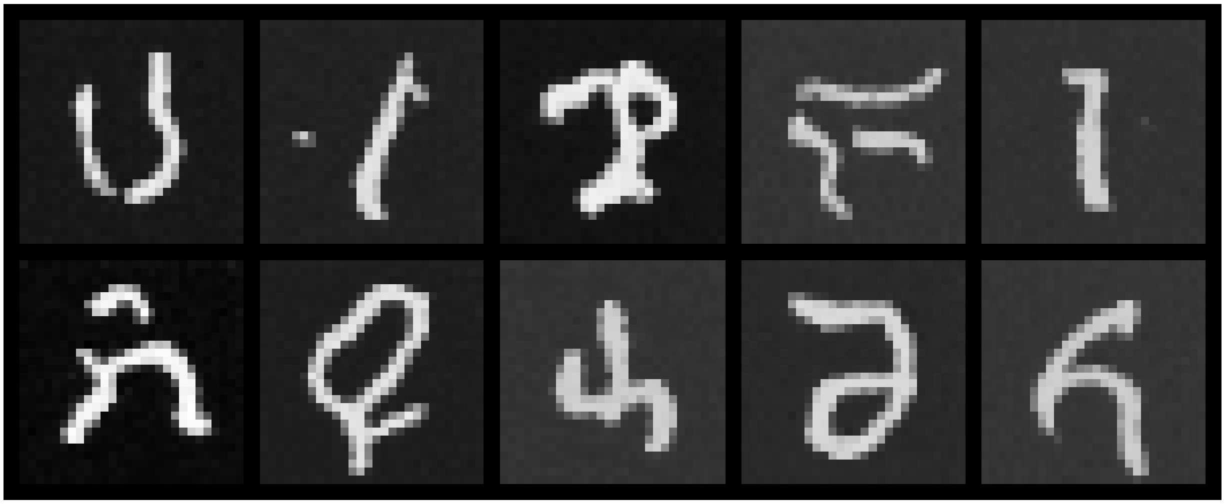

```
Epoch 7/30
--------------------
  Step 0/750, Loss: 0.0258
  Step 100/750, Loss: 0.0305
  Step 200/750, Loss: 0.0316
  Step 300/750, Loss: 0.0247
  Step 400/750, Loss: 0.0246
  Step 500/750, Loss: 0.0215
  Generating samples...
```




```
  Step 600/750, Loss: 0.0254
  Step 700/750, Loss: 0.0272

Training - Epoch 7 average loss: 0.0266
Running validation...
Validation - Epoch 7 average loss: 0.0275
Learning rate: 0.001000
```

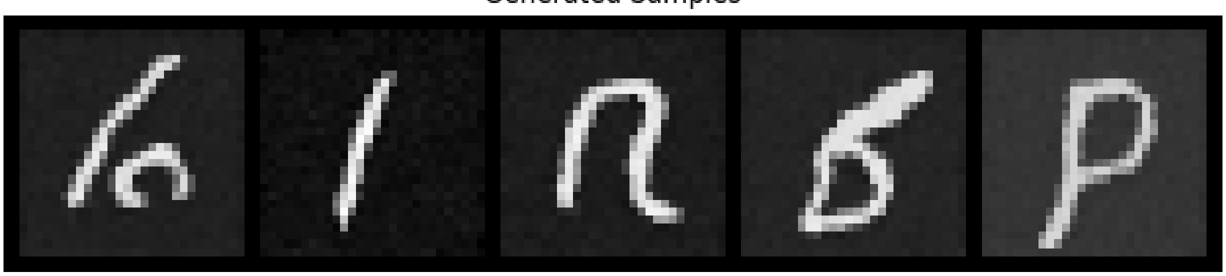
```
Generating samples for visual progress check...
```

Generated Samples



```
No improvement for 1/10 epochs

Epoch 8/30
--------------------
  Step 0/750, Loss: 0.0264
  Step 100/750, Loss: 0.0256
  Step 200/750, Loss: 0.0304
  Step 300/750, Loss: 0.0325
  Step 400/750, Loss: 0.0243
  Step 500/750, Loss: 0.0292
  Generating samples...
```

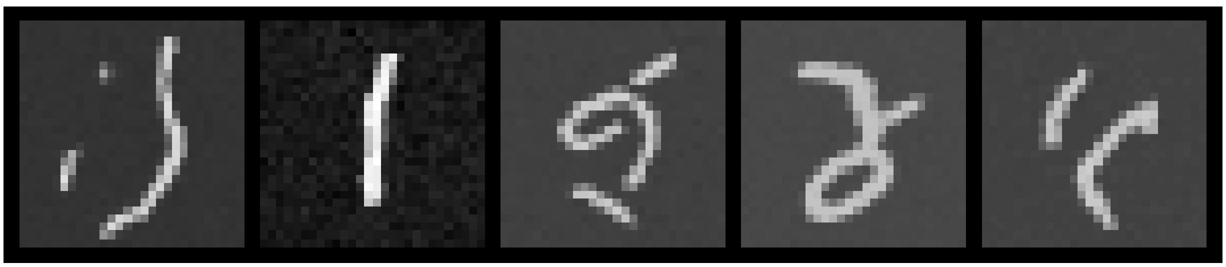
Generated Samples



```
  Step 600/750, Loss: 0.0238
  Step 700/750, Loss: 0.0229

Training - Epoch 8 average loss: 0.0261
Running validation...
Validation - Epoch 8 average loss: 0.0261
```

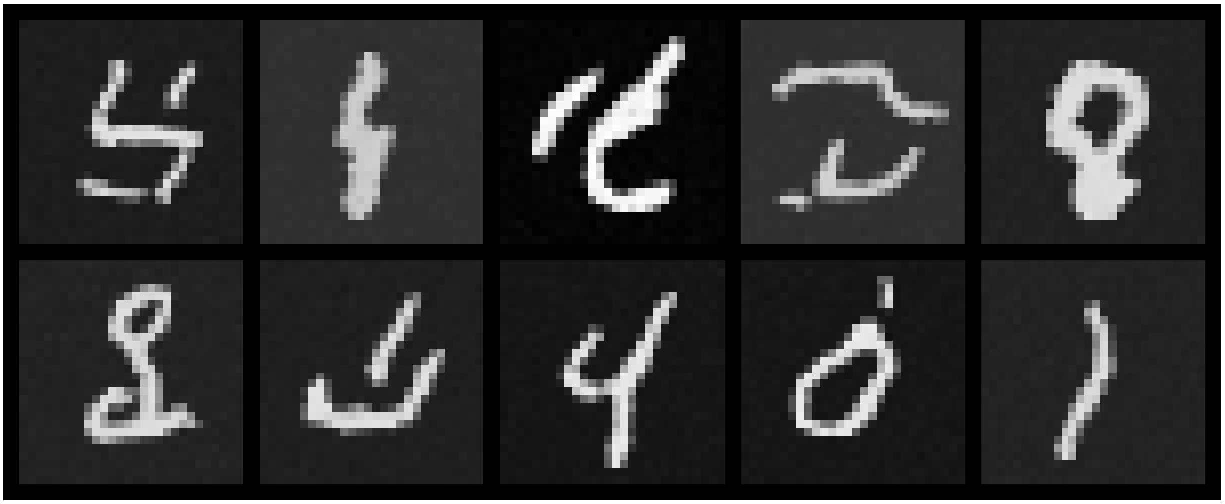
```
Learning rate: 0.001000
Created backup at best_diffusion_model.pt.backup
Model successfully saved to best_diffusion_model.pt
✓ New best model saved! (Val Loss: 0.0261)

Epoch 9/30
--------------------
  Step 0/750, Loss: 0.0289
  Step 100/750, Loss: 0.0254
  Step 200/750, Loss: 0.0245
  Step 300/750, Loss: 0.0239
  Step 400/750, Loss: 0.0300
  Step 500/750, Loss: 0.0258
  Generating samples...
```

Generated Samples



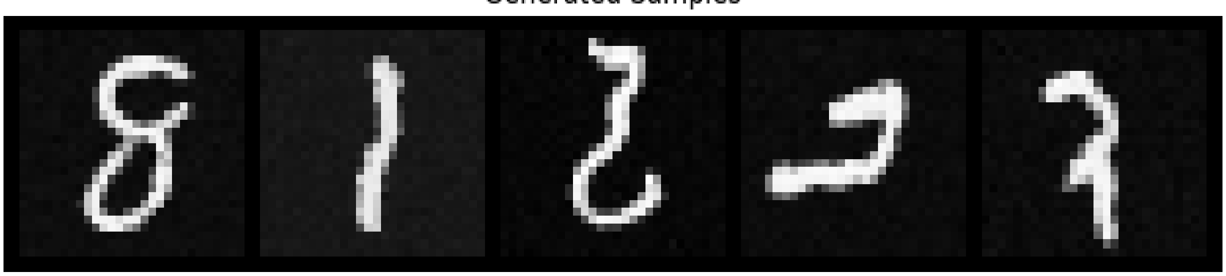
```
  Step 600/750, Loss: 0.0281
  Step 700/750, Loss: 0.0255

Training - Epoch 9 average loss: 0.0259
Running validation...
Validation - Epoch 9 average loss: 0.0261
Learning rate: 0.001000

Generating samples for visual progress check...
```

Generated Samples



```
Created backup at best_diffusion_model.pt.backup
Model successfully saved to best_diffusion_model.pt
✓ New best model saved! (Val Loss: 0.0261)

Epoch 10/30
--------------------
  Step 0/750, Loss: 0.0277
  Step 100/750, Loss: 0.0223
  Step 200/750, Loss: 0.0325
  Step 300/750, Loss: 0.0204
  Step 400/750, Loss: 0.0198
  Step 500/750, Loss: 0.0251
  Generating samples...
```

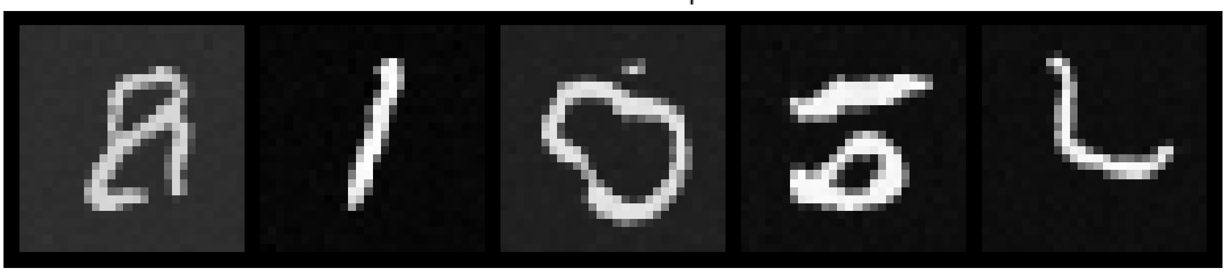
Generated Samples



```
  Step 600/750, Loss: 0.0229
  Step 700/750, Loss: 0.0253

Training - Epoch 10 average loss: 0.0259
Running validation...
Validation - Epoch 10 average loss: 0.0251
Learning rate: 0.001000
Created backup at best_diffusion_model.pt.backup
```

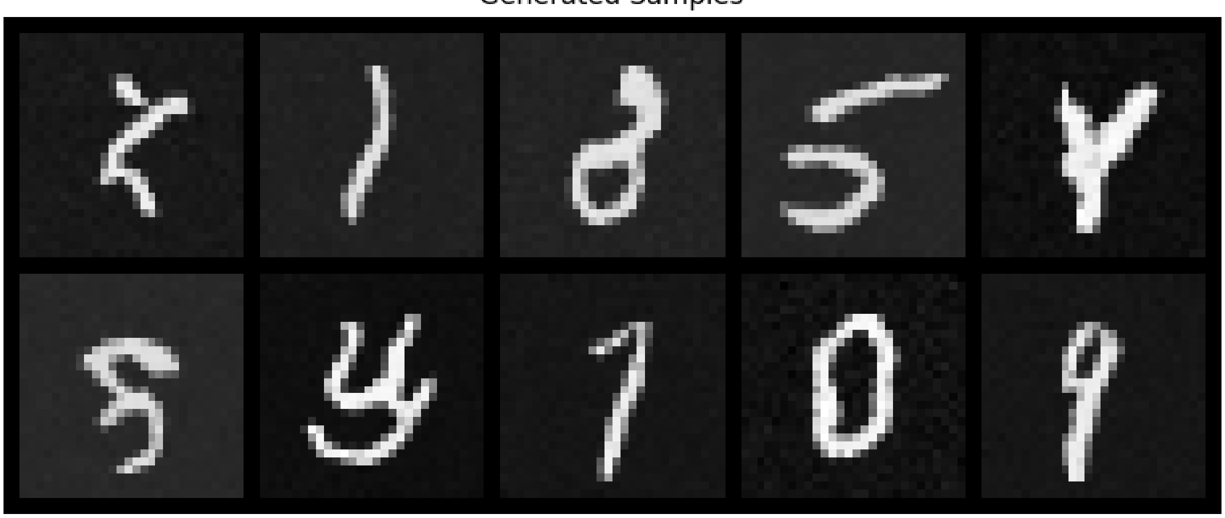
```
Model successfully saved to best_diffusion_model.pt
✓ New best model saved! (Val Loss: 0.0251)

Epoch 11/30
--------------------
  Step 0/750, Loss: 0.0308
  Step 100/750, Loss: 0.0232
  Step 200/750, Loss: 0.0297
  Step 300/750, Loss: 0.0204
  Step 400/750, Loss: 0.0304
  Step 500/750, Loss: 0.0205
  Generating samples...
```

Generated Samples



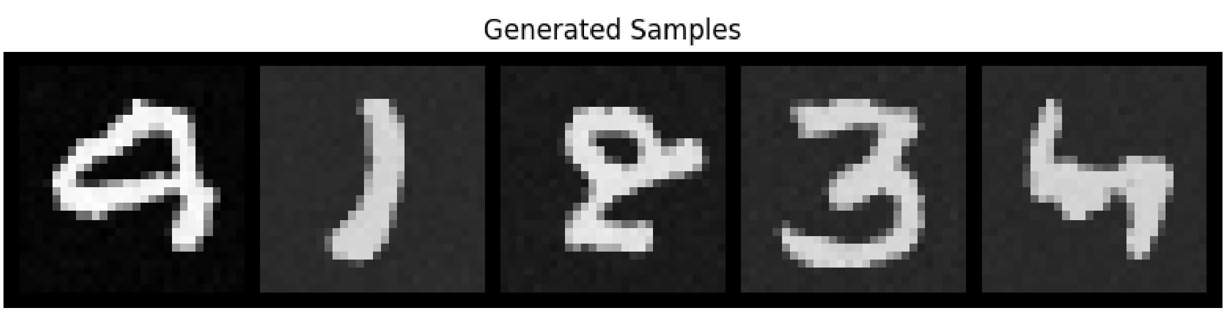
```
  Step 600/750, Loss: 0.0318
  Step 700/750, Loss: 0.0281

Training - Epoch 11 average loss: 0.0258
Running validation...
Validation - Epoch 11 average loss: 0.0257
Learning rate: 0.001000

Generating samples for visual progress check...
```

Generated Samples



```
No improvement for 1/10 epochs
```

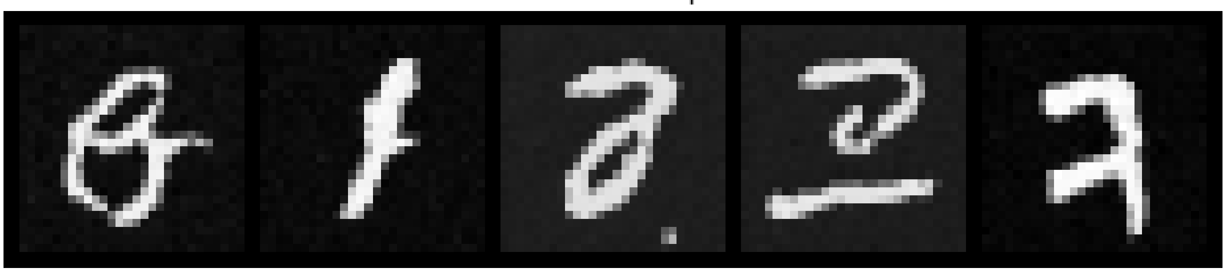

```
Epoch 12/30
--------------------
  Step 0/750, Loss: 0.0267
  Step 100/750, Loss: 0.0230
  Step 200/750, Loss: 0.0245
  Step 300/750, Loss: 0.0274
  Step 400/750, Loss: 0.0201
  Step 500/750, Loss: 0.0326
  Generating samples...
```




```
  Step 600/750, Loss: 0.0247
  Step 700/750, Loss: 0.0198

Training - Epoch 12 average loss: 0.0257
Running validation...
```

```
Validation - Epoch 12 average loss: 0.0260
Learning rate: 0.001000
No improvement for 2/10 epochs

Epoch 13/30
--------------------
  Step 0/750, Loss: 0.0260
  Step 100/750, Loss: 0.0273
  Step 200/750, Loss: 0.0286
  Step 300/750, Loss: 0.0247
  Step 400/750, Loss: 0.0257
  Step 500/750, Loss: 0.0257
  Generating samples...
```

Generated Samples

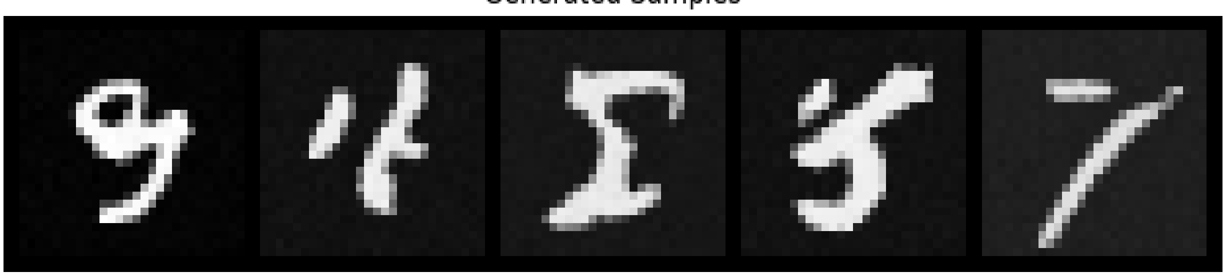


```
  Step 600/750, Loss: 0.0235
  Step 700/750, Loss: 0.0258

Training - Epoch 13 average loss: 0.0254
Running validation...
Validation - Epoch 13 average loss: 0.0256
Learning rate: 0.001000

Generating samples for visual progress check...
```

Generated Samples

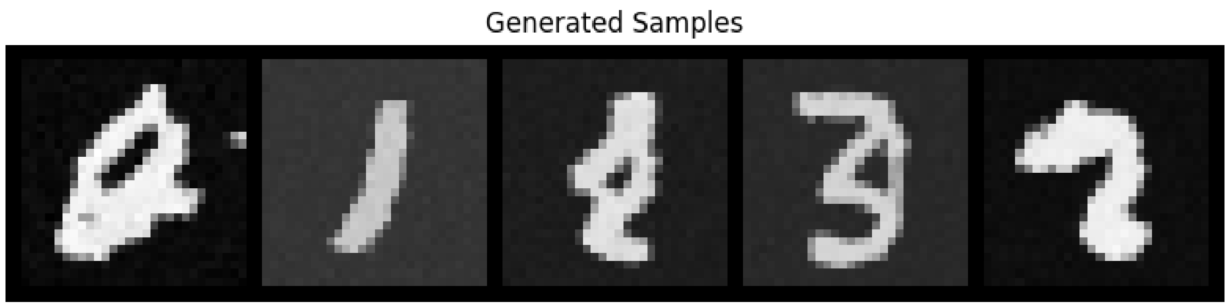
```
No improvement for 3/10 epochs

Epoch 14/30
--------------------
  Step 0/750, Loss: 0.0299
  Step 100/750, Loss: 0.0328
  Step 200/750, Loss: 0.0234
  Step 300/750, Loss: 0.0310
  Step 400/750, Loss: 0.0233
  Step 500/750, Loss: 0.0243
  Generating samples...
```

Generated Samples

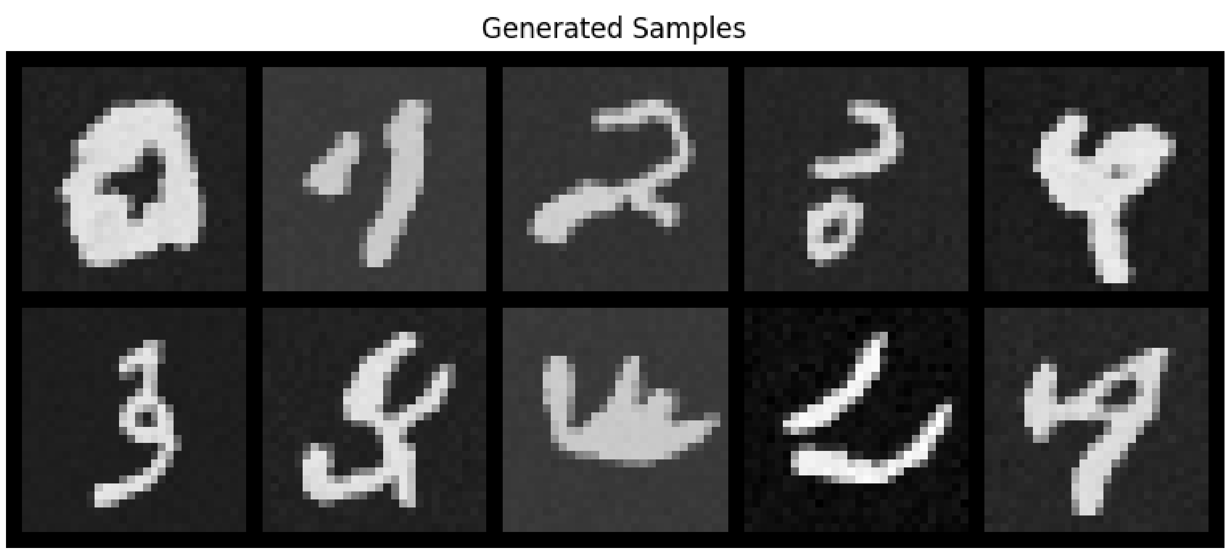


```
  Step 600/750, Loss: 0.0171
  Step 700/750, Loss: 0.0252

Training - Epoch 14 average loss: 0.0252
Running validation...
Validation - Epoch 14 average loss: 0.0252
Learning rate: 0.001000
No improvement for 4/10 epochs

Epoch 15/30
```

```
--------------------
  Step 0/750, Loss: 0.0258
  Step 100/750, Loss: 0.0202
  Step 200/750, Loss: 0.0207
  Step 300/750, Loss: 0.0232
  Step 400/750, Loss: 0.0209
  Step 500/750, Loss: 0.0236
  Generating samples...
```



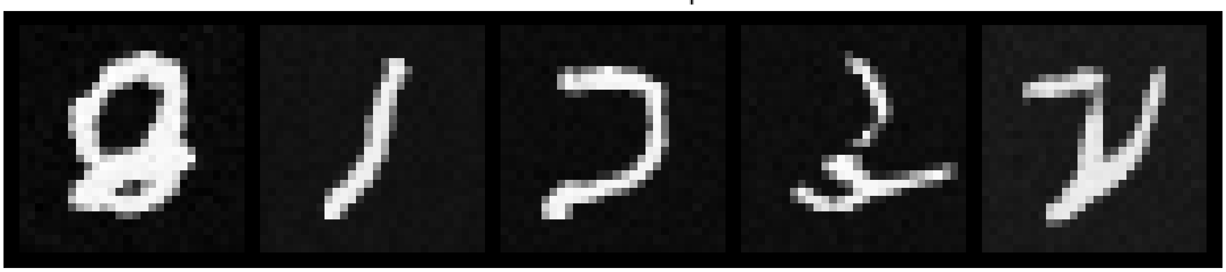

```
  Step 600/750, Loss: 0.0293
  Step 700/750, Loss: 0.0256

Training - Epoch 15 average loss: 0.0252
Running validation...
Validation - Epoch 15 average loss: 0.0243
Learning rate: 0.001000

Generating samples for visual progress check...
```




```
Created backup at best_diffusion_model.pt.backup
Model successfully saved to best_diffusion_model.pt
```

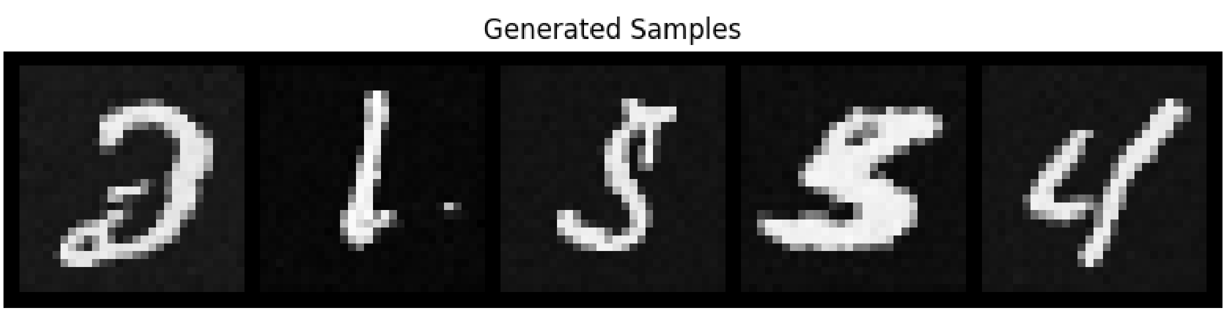
```
✓ New best model saved! (Val Loss: 0.0243)

Epoch 16/30
--------------------
  Step 0/750, Loss: 0.0259
  Step 100/750, Loss: 0.0261
  Step 200/750, Loss: 0.0230
  Step 300/750, Loss: 0.0280
  Step 400/750, Loss: 0.0218
  Step 500/750, Loss: 0.0287
  Generating samples...
```

Generated Samples



```
  Step 600/750, Loss: 0.0188
  Step 700/750, Loss: 0.0228

Training - Epoch 16 average loss: 0.0250
Running validation...
Validation - Epoch 16 average loss: 0.0245
Learning rate: 0.001000
No improvement for 1/10 epochs
```

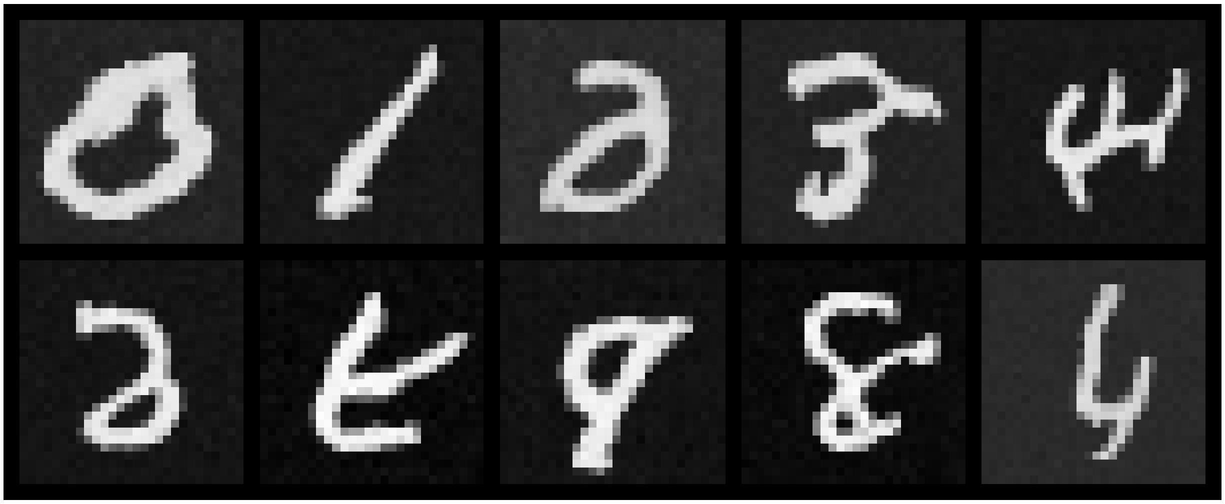

```
Epoch 17/30
--------------------
  Step 0/750, Loss: 0.0201
  Step 100/750, Loss: 0.0247
  Step 200/750, Loss: 0.0247
  Step 300/750, Loss: 0.0227
  Step 400/750, Loss: 0.0244
  Step 500/750, Loss: 0.0342
  Generating samples...
```




```
  Step 600/750, Loss: 0.0329
  Step 700/750, Loss: 0.0239

Training - Epoch 17 average loss: 0.0248
Running validation...
Validation - Epoch 17 average loss: 0.0251
Learning rate: 0.001000
```

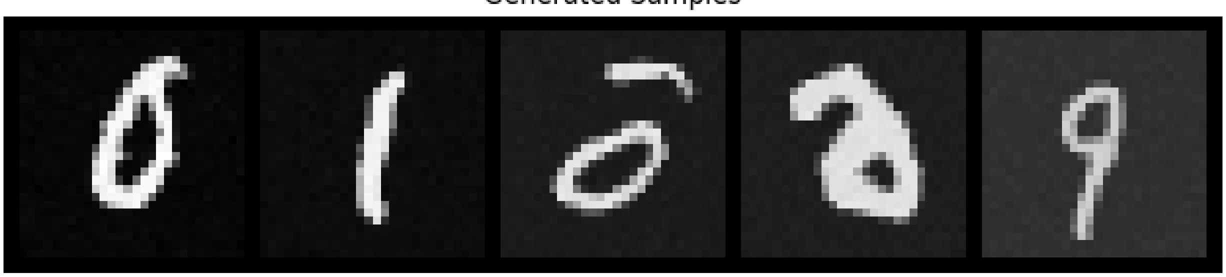
```
Generating samples for visual progress check...
```

Generated Samples



```
No improvement for 2/10 epochs

Epoch 18/30
--------------------
  Step 0/750, Loss: 0.0202
  Step 100/750, Loss: 0.0228
  Step 200/750, Loss: 0.0223
  Step 300/750, Loss: 0.0223
  Step 400/750, Loss: 0.0211
  Step 500/750, Loss: 0.0173
  Generating samples...
```

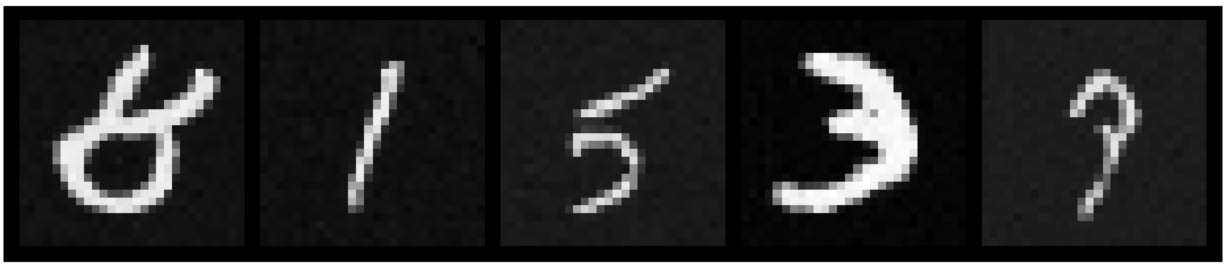
Generated Samples



```
  Step 600/750, Loss: 0.0290
  Step 700/750, Loss: 0.0256

Training - Epoch 18 average loss: 0.0246
Running validation...
Validation - Epoch 18 average loss: 0.0244
```

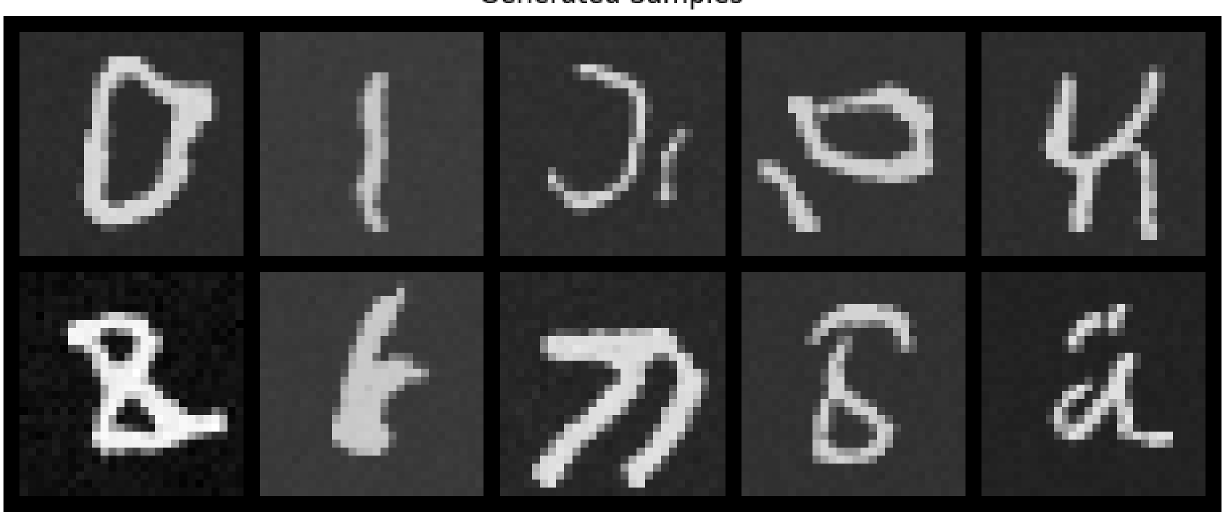
```
Learning rate: 0.001000
No improvement for 3/10 epochs

Epoch 19/30
--------------------
  Step 0/750, Loss: 0.0258
  Step 100/750, Loss: 0.0267
  Step 200/750, Loss: 0.0249
  Step 300/750, Loss: 0.0207
  Step 400/750, Loss: 0.0216
  Step 500/750, Loss: 0.0280
  Generating samples...
```

Generated Samples



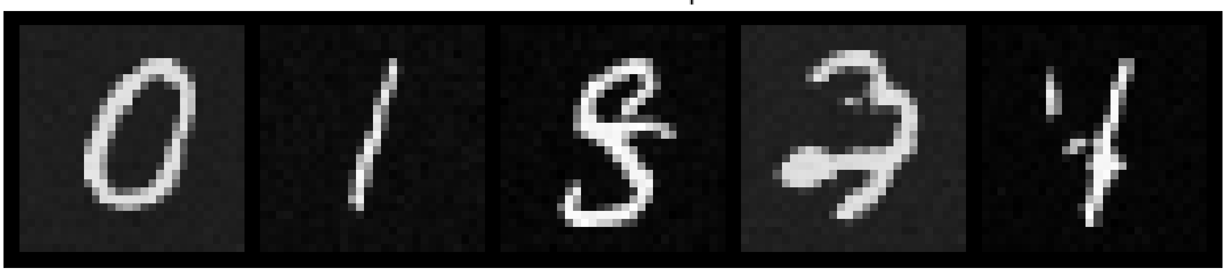
```
  Step 600/750, Loss: 0.0207
  Step 700/750, Loss: 0.0293

Training - Epoch 19 average loss: 0.0249
Running validation...
Validation - Epoch 19 average loss: 0.0241
Learning rate: 0.001000

Generating samples for visual progress check...
```

Generated Samples

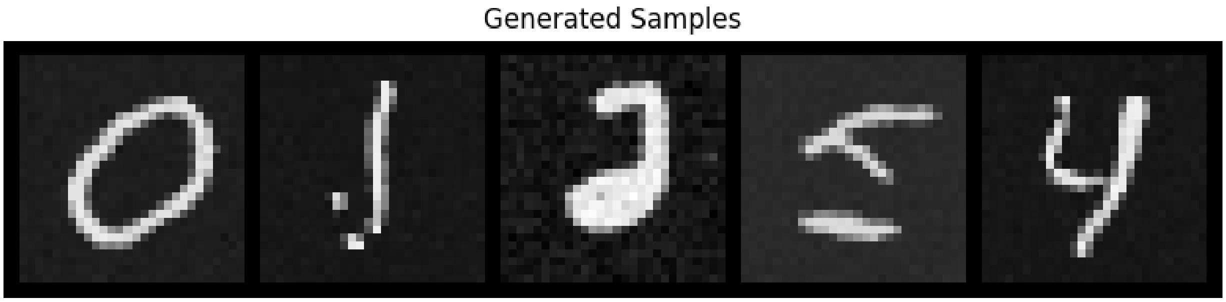


```
Created backup at best_diffusion_model.pt.backup
Model successfully saved to best_diffusion_model.pt
✓ New best model saved! (Val Loss: 0.0241)

Epoch 20/30
--------------------
  Step 0/750, Loss: 0.0198
  Step 100/750, Loss: 0.0245
  Step 200/750, Loss: 0.0168
  Step 300/750, Loss: 0.0205
  Step 400/750, Loss: 0.0276
  Step 500/750, Loss: 0.0269
  Generating samples...
```

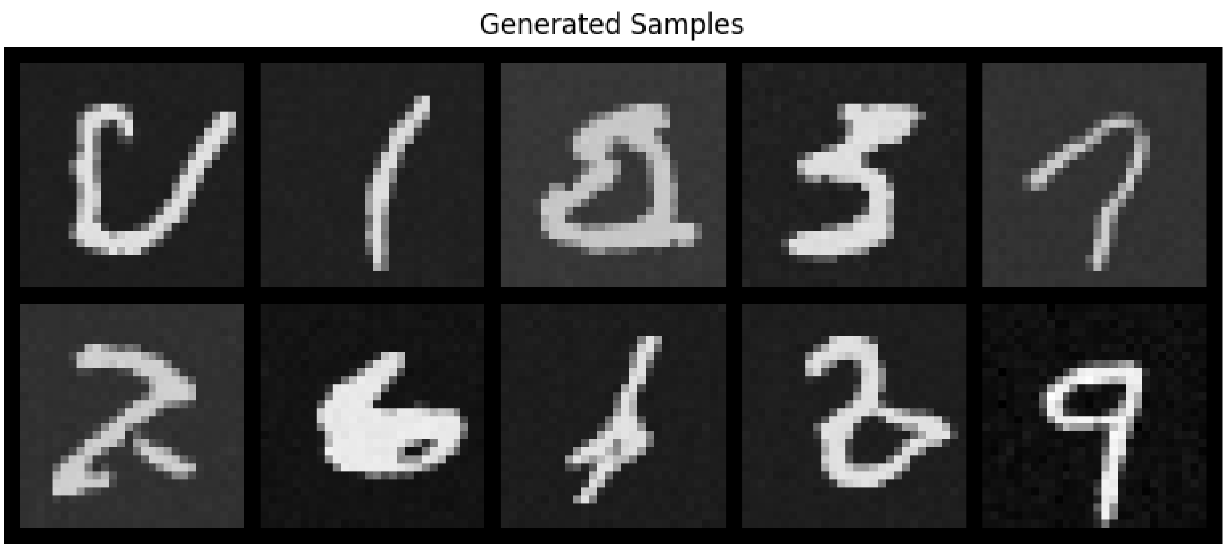
Generated Samples



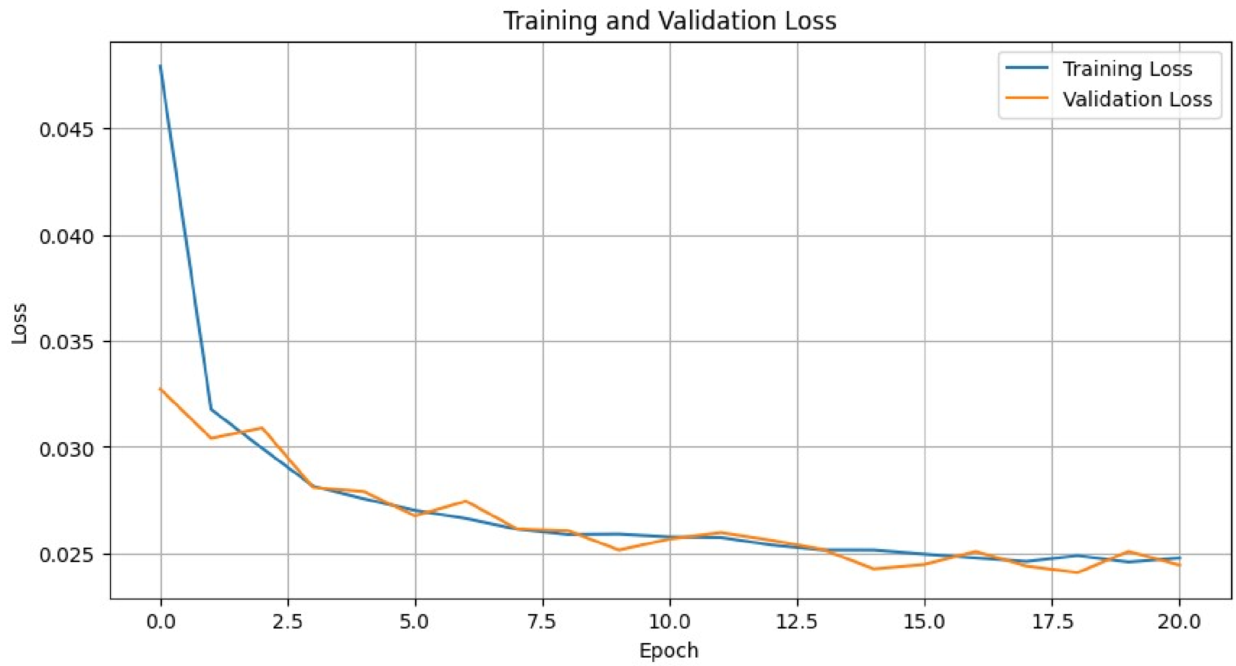
```
  Step 600/750, Loss: 0.0255
  Step 700/750, Loss: 0.0135

Training - Epoch 20 average loss: 0.0246
Running validation...
Validation - Epoch 20 average loss: 0.0251
Learning rate: 0.001000
No improvement for 1/10 epochs

Epoch 21/30
--------------------
  Step 0/750, Loss: 0.0211
  Step 100/750, Loss: 0.0202
  Step 200/750, Loss: 0.0234
  Step 300/750, Loss: 0.0218
  Step 400/750, Loss: 0.0283
  Step 500/750, Loss: 0.0299
  Generating samples...
```



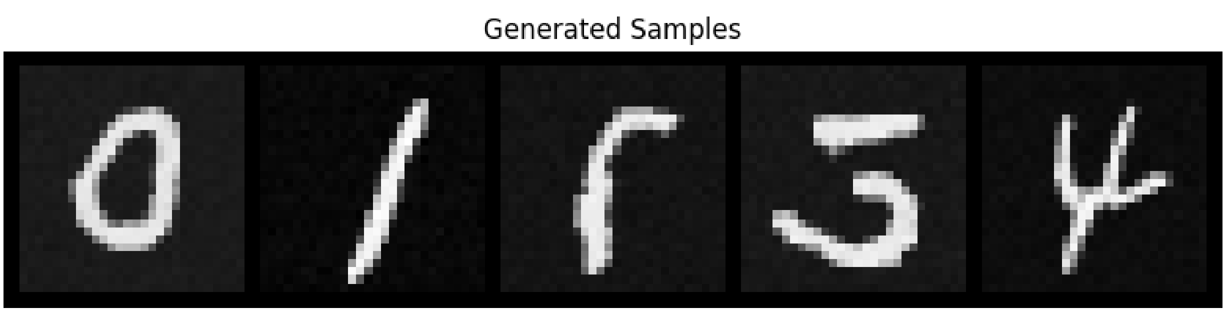

```
  Step 600/750, Loss: 0.0212
  Step 700/750, Loss: 0.0205

Training - Epoch 21 average loss: 0.0248
Running validation...
Validation - Epoch 21 average loss: 0.0244
Learning rate: 0.001000

Generating samples for visual progress check...
```




```
No improvement for 2/10 epochs
```



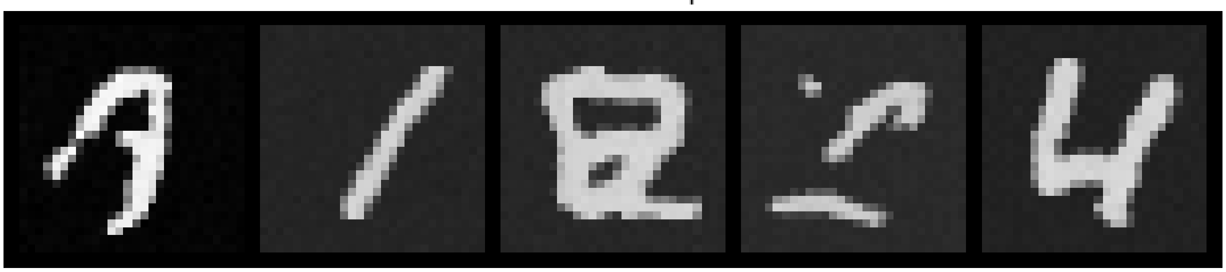

```
Epoch 22/30
--------------------
  Step 0/750, Loss: 0.0220
  Step 100/750, Loss: 0.0252
  Step 200/750, Loss: 0.0299
  Step 300/750, Loss: 0.0232
  Step 400/750, Loss: 0.0174
  Step 500/750, Loss: 0.0192
  Generating samples...
```




```
  Step 600/750, Loss: 0.0256
  Step 700/750, Loss: 0.0274

Training - Epoch 22 average loss: 0.0244
Running validation...
Validation - Epoch 22 average loss: 0.0257
Learning rate: 0.001000
```

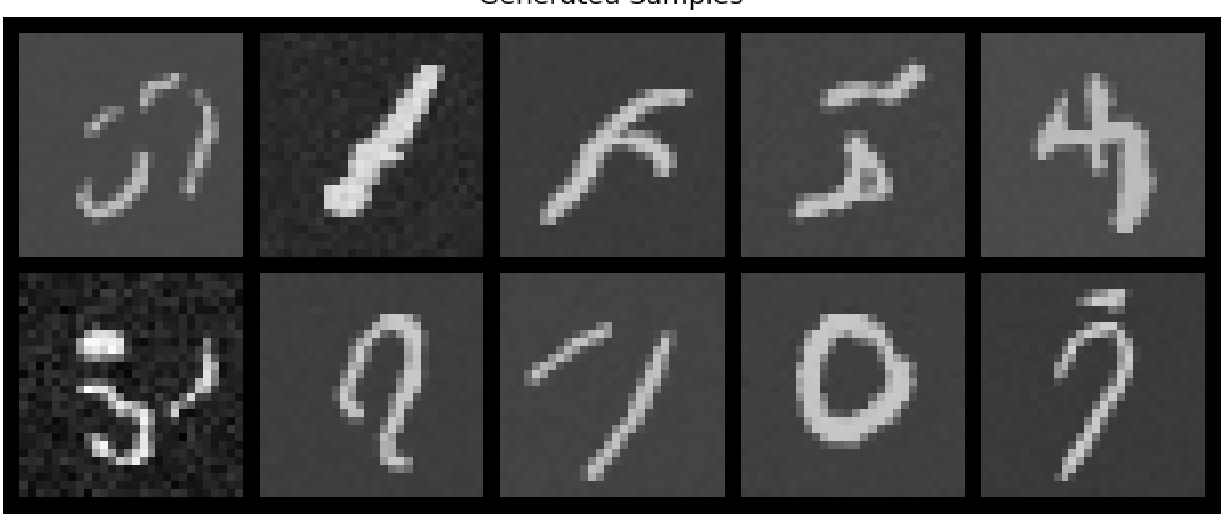
```
No improvement for 3/10 epochs

Epoch 23/30
--------------------
  Step 0/750, Loss: 0.0335
  Step 100/750, Loss: 0.0256
  Step 200/750, Loss: 0.0198
  Step 300/750, Loss: 0.0207
  Step 400/750, Loss: 0.0273
  Step 500/750, Loss: 0.0220
  Generating samples...
```

Generated Samples



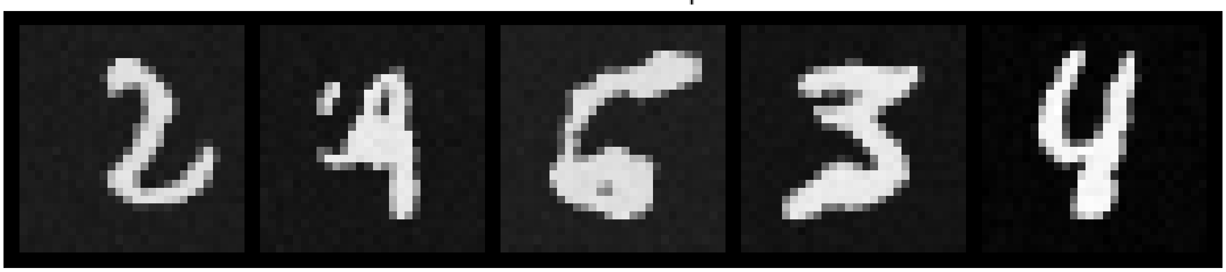
```
  Step 600/750, Loss: 0.0236
  Step 700/750, Loss: 0.0205

Training - Epoch 23 average loss: 0.0245
Running validation...
Validation - Epoch 23 average loss: 0.0242
Learning rate: 0.001000

Generating samples for visual progress check...
```

Generated Samples



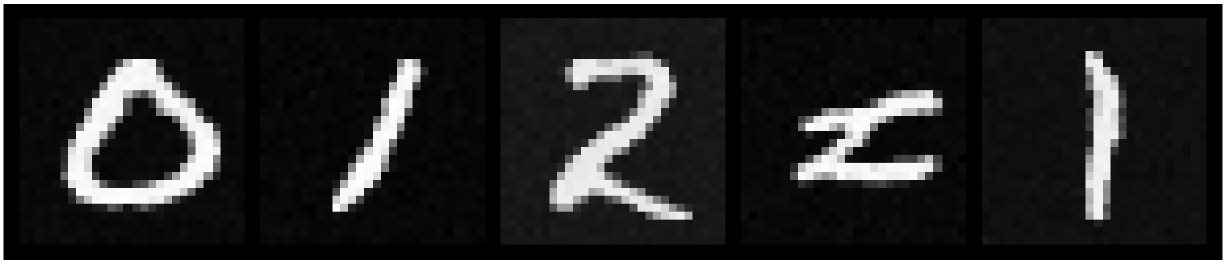
```
No improvement for 4/10 epochs

Epoch 24/30
--------------------
  Step 0/750, Loss: 0.0334
  Step 100/750, Loss: 0.0201
  Step 200/750, Loss: 0.0228
  Step 300/750, Loss: 0.0209
  Step 400/750, Loss: 0.0296
  Step 500/750, Loss: 0.0268
  Generating samples...
```

Generated Samples



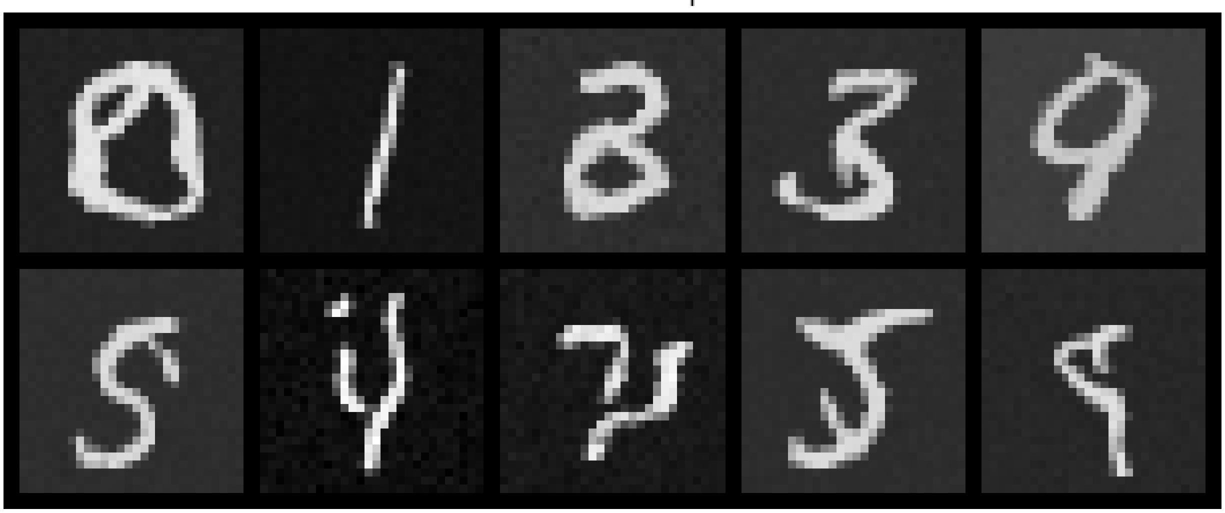
```
  Step 600/750, Loss: 0.0230
  Step 700/750, Loss: 0.0268

Training - Epoch 24 average loss: 0.0247
Running validation...
Validation - Epoch 24 average loss: 0.0243
Learning rate: 0.001000
No improvement for 5/10 epochs

Epoch 25/30
--------------------
  Step 0/750, Loss: 0.0208
  Step 100/750, Loss: 0.0220
  Step 200/750, Loss: 0.0302
  Step 300/750, Loss: 0.0283
  Step 400/750, Loss: 0.0236
  Step 500/750, Loss: 0.0289
  Generating samples...
```

Generated Samples

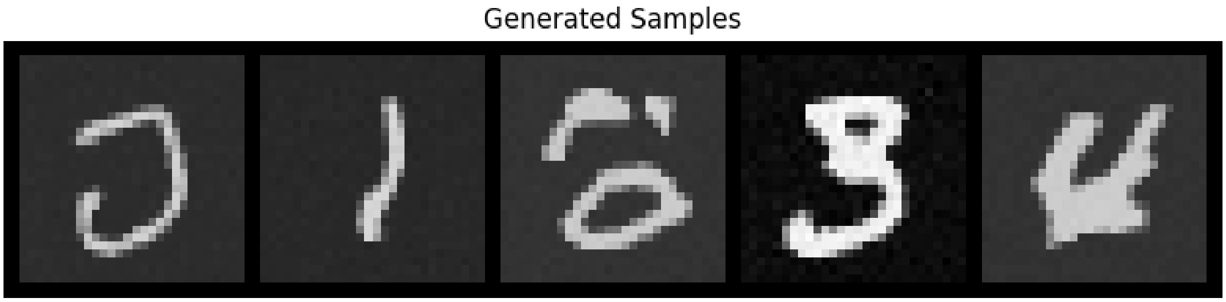


```
  Step 600/750, Loss: 0.0253
  Step 700/750, Loss: 0.0259

Training - Epoch 25 average loss: 0.0246
Running validation...
Validation - Epoch 25 average loss: 0.0246
Learning rate: 0.000500

Generating samples for visual progress check...
```

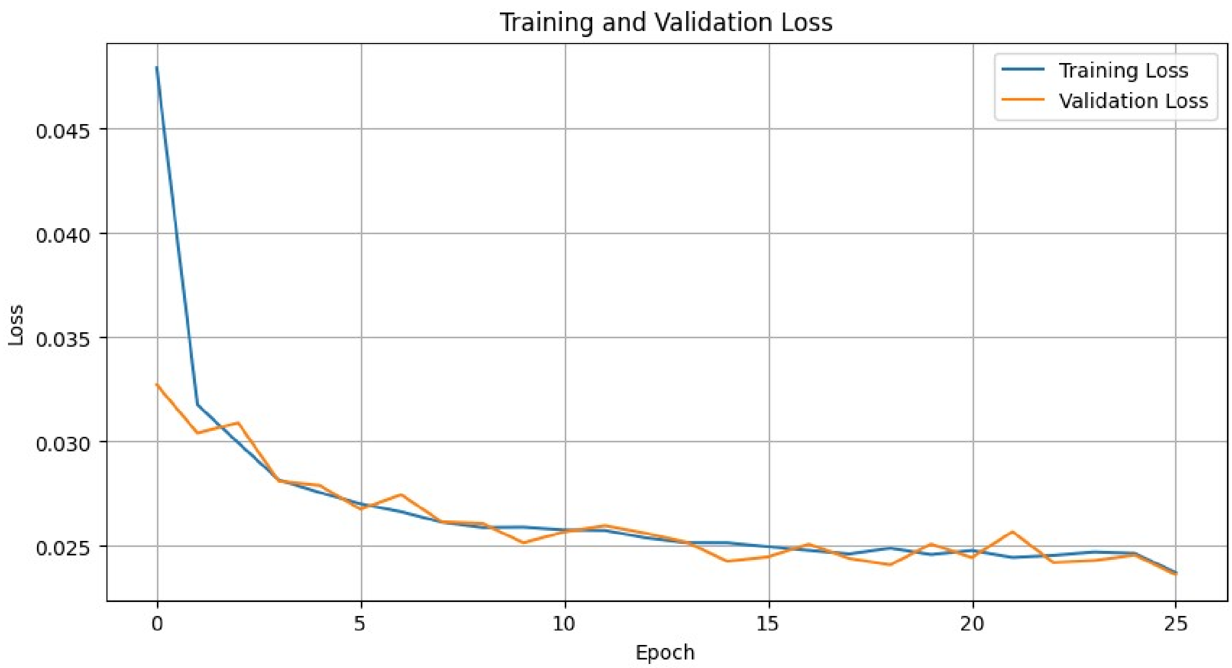
Generated Samples



```
No improvement for 6/10 epochs

Epoch 26/30
--------------------
  Step 0/750, Loss: 0.0224
  Step 100/750, Loss: 0.0188
  Step 200/750, Loss: 0.0255
  Step 300/750, Loss: 0.0273
  Step 400/750, Loss: 0.0187
  Step 500/750, Loss: 0.0224
  Generating samples...
```



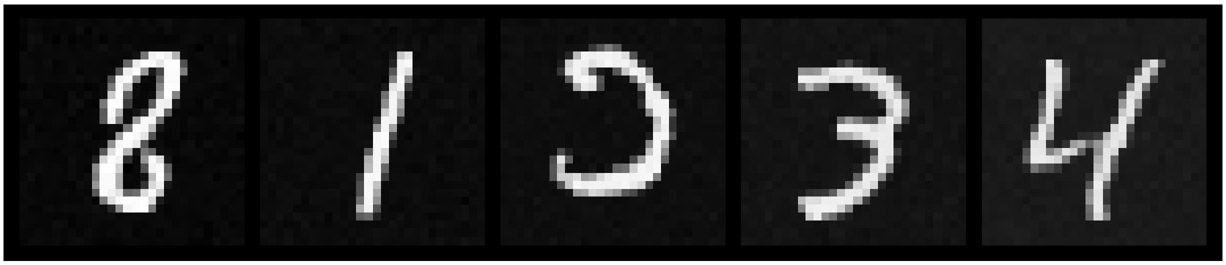

```
  Step 600/750, Loss: 0.0295
  Step 700/750, Loss: 0.0204

Training - Epoch 26 average loss: 0.0237
Running validation...
Validation - Epoch 26 average loss: 0.0236
Learning rate: 0.000500
Created backup at best_diffusion_model.pt.backup
Model successfully saved to best_diffusion_model.pt
✓ New best model saved! (Val Loss: 0.0236)
```




```
Epoch 27/30
--------------------
  Step 0/750, Loss: 0.0255
  Step 100/750, Loss: 0.0249
  Step 200/750, Loss: 0.0251
  Step 300/750, Loss: 0.0284
```

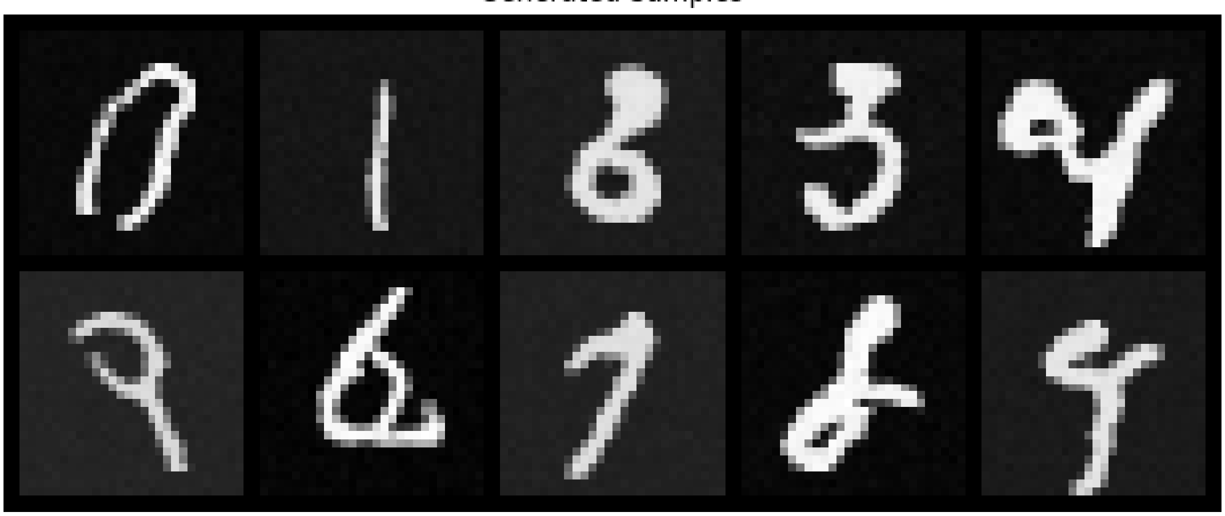
```
  Step 400/750, Loss: 0.0224
  Step 500/750, Loss: 0.0202
  Generating samples...
```

Generated Samples



```
  Step 600/750, Loss: 0.0276
  Step 700/750, Loss: 0.0245

Training - Epoch 27 average loss: 0.0235
Running validation...
Validation - Epoch 27 average loss: 0.0236
Learning rate: 0.000500

Generating samples for visual progress check...
```

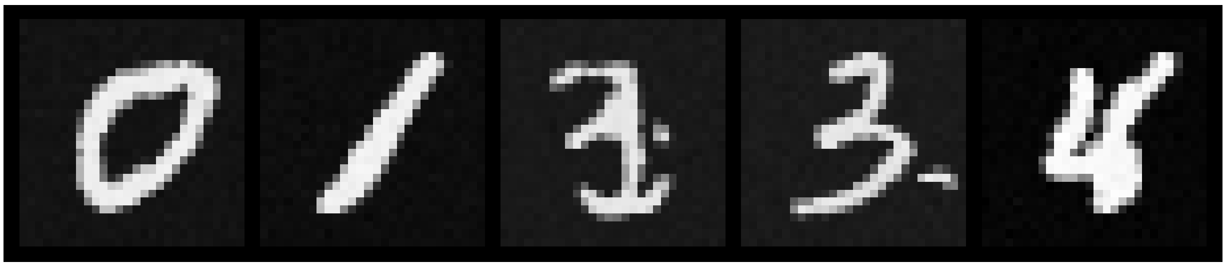
Generated Samples



```
Created backup at best_diffusion_model.pt.backup
Model successfully saved to best_diffusion_model.pt
✓ New best model saved! (Val Loss: 0.0236)

Epoch 28/30
--------------------
  Step 0/750, Loss: 0.0257
```

```
  Step 100/750, Loss: 0.0250
  Step 200/750, Loss: 0.0220
  Step 300/750, Loss: 0.0247
  Step 400/750, Loss: 0.0192
  Step 500/750, Loss: 0.0223
  Generating samples...
```

Generated Samples

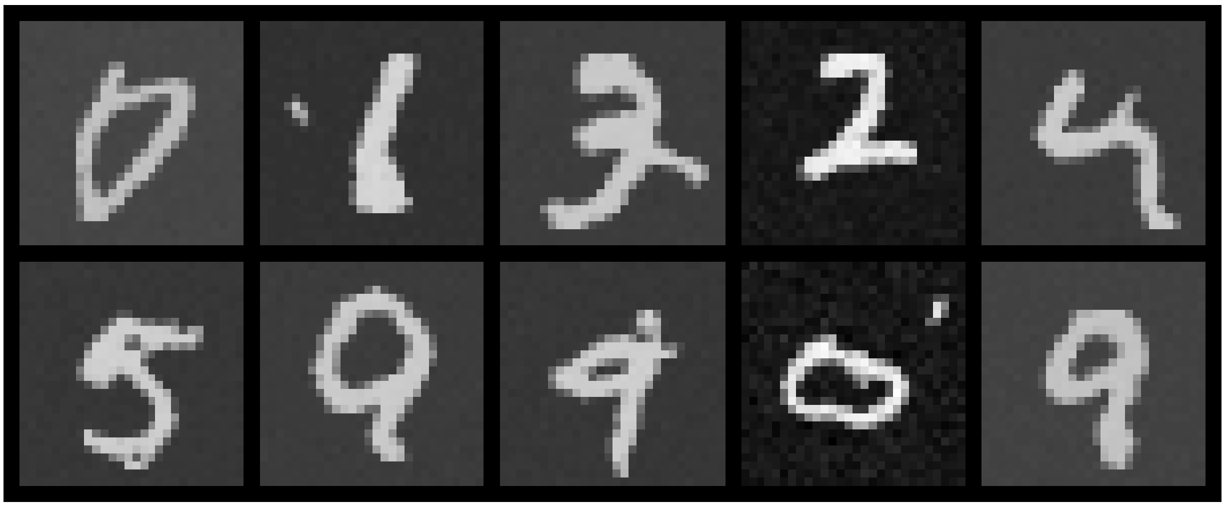


```
  Step 600/750, Loss: 0.0203
  Step 700/750, Loss: 0.0209

Training - Epoch 28 average loss: 0.0238
Running validation...
Validation - Epoch 28 average loss: 0.0235
Learning rate: 0.000500
Created backup at best_diffusion_model.pt.backup
Model successfully saved to best_diffusion_model.pt
✓ New best model saved! (Val Loss: 0.0235)

Epoch 29/30
--------------------
  Step 0/750, Loss: 0.0247
  Step 100/750, Loss: 0.0210
  Step 200/750, Loss: 0.0262
  Step 300/750, Loss: 0.0275
  Step 400/750, Loss: 0.0216
  Step 500/750, Loss: 0.0232
  Generating samples...
```

Generated Samples

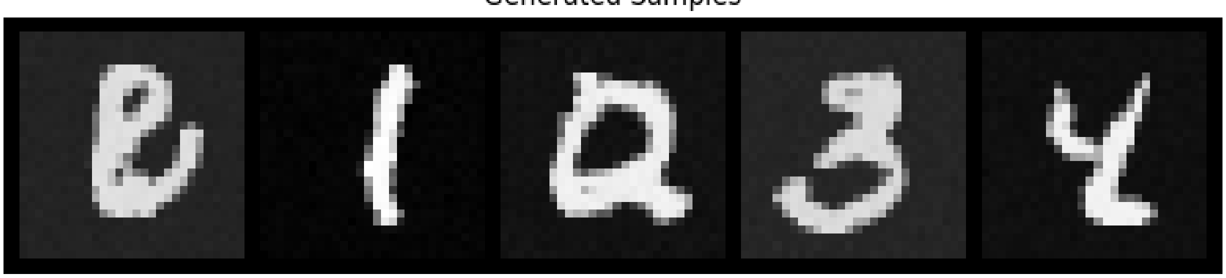
```
  Step 600/750, Loss: 0.0228
  Step 700/750, Loss: 0.0232

Training - Epoch 29 average loss: 0.0238
Running validation...
Validation - Epoch 29 average loss: 0.0238
Learning rate: 0.000500

Generating samples for visual progress check...
```

Generated Samples

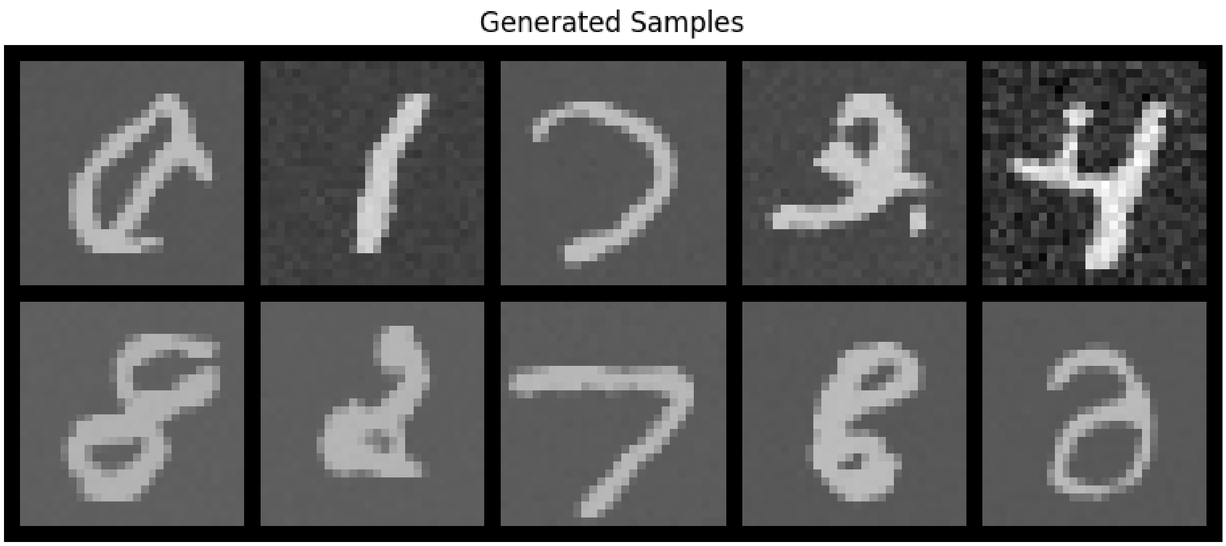


```
No improvement for 1/10 epochs

Epoch 30/30
--------------------
  Step 0/750, Loss: 0.0266
  Step 100/750, Loss: 0.0230
  Step 200/750, Loss: 0.0243
  Step 300/750, Loss: 0.0238
  Step 400/750, Loss: 0.0200
  Step 500/750, Loss: 0.0271
  Generating samples...
```

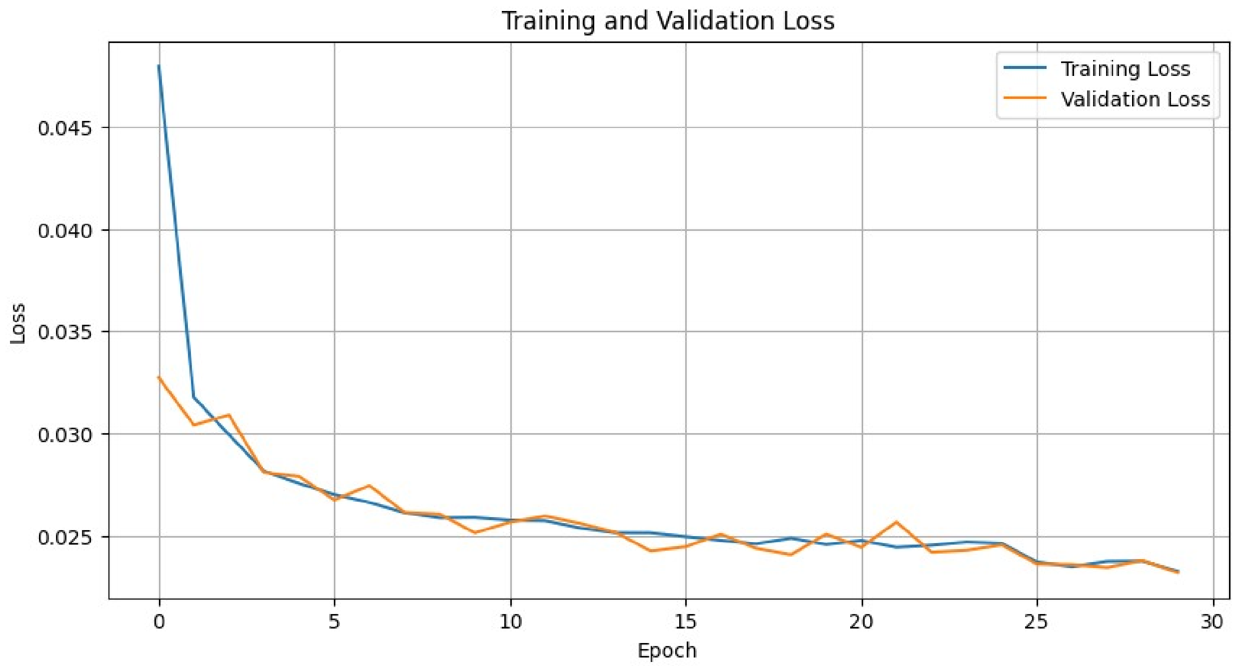
Generated Samples



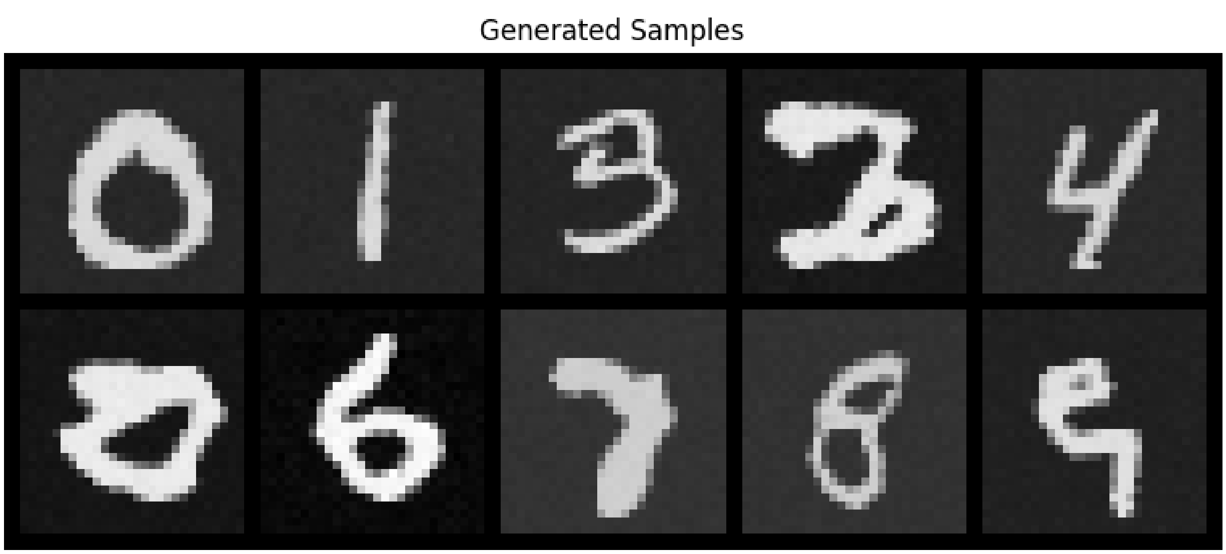
```
  Step 600/750, Loss: 0.0228
  Step 700/750, Loss: 0.0187

Training - Epoch 30 average loss: 0.0233
Running validation...
Validation - Epoch 30 average loss: 0.0232
Learning rate: 0.000500

Generating samples for visual progress check...
```

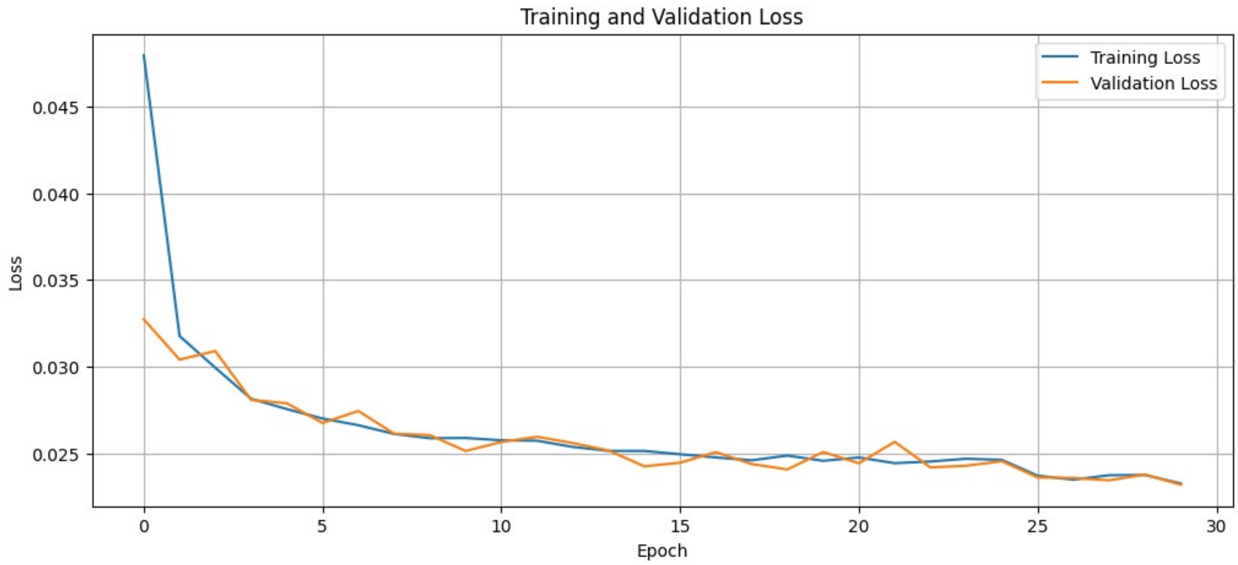



```
Created backup at best_diffusion_model.pt.backup
Model successfully saved to best_diffusion_model.pt
✓ New best model saved! (Val Loss: 0.0232)
```




```
==================================================
TRAINING COMPLETE
==================================================
Best validation loss: 0.0232
Generating final samples...
```






```
# 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,
```

```python
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.")

import torch.nn.functional as F

def train_step(x, c):
    """
    Performs a single training step for the diffusion model.

    Args:
        x (torch.Tensor): Batch of clean images [batch_size, channels,
```

```python
height, width]
        c (torch.Tensor): Batch of class labels [batch_size]

    Returns:
        torch.Tensor: Mean squared error loss value
    """
    x = x.to(device)
    c = c.to(device)

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

    # Conditioning mask (all ones; can be modified for classifier-free
guidance)
    c_mask = torch.ones((x.shape[0], 1), device=device)

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

    # Add noise according to forward diffusion
    noise = torch.randn_like(x)
    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)
    x_t = sqrt_alpha_bar_t * x + sqrt_one_minus_alpha_bar_t * noise

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

    # Compute mean squared error loss between true noise and predicted
noise
    loss = F.mse_loss(predicted_noise, noise)

    return loss


# Training configuration
early_stopping_patience = 10  # epochs without improvement before
stopping
gradient_clip_value = 1.0     # max gradient norm for stability
display_frequency = 100       # steps interval for printing loss
generate_frequency = 500      # steps interval for generating samples
EPOCHS = 8                  # set your desired number of epochs

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

print("\n" + "="*50)
```

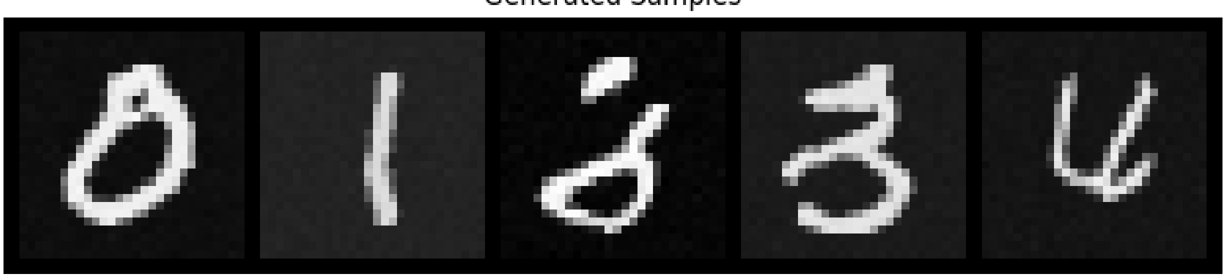
```
print("STARTING TRAINING")
print("="*50)

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

        model.train()
        epoch_losses = []

        for step, (images, labels) in enumerate(train_loader):  # Make
sure your train DataLoader is named train_loader
            images = images.to(device)
            labels = labels.to(device)

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

            # Gradient clipping
            torch.nn.utils.clip_grad_norm_(model.parameters(),
max_norm=gradient_clip_value)
            optimizer.step()

            epoch_losses.append(loss.item())

            if step % display_frequency == 0:
                print(f"  Step {step}/{len(train_loader)}, Loss:
{loss.item():.4f}")
                if step % generate_frequency == 0 and step > 0:
                    print("  Generating samples...")
                    generate_samples(model, n_samples=5)  # define
generate_samples() yourself

        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}")

        model.eval()
        val_epoch_losses = []
        print("Running validation...")
        with torch.no_grad():
            for val_images, val_labels in val_loader:  # Make sure
your validation DataLoader is val_loader
                val_images = val_images.to(device)
                val_labels = val_labels.to(device)
                val_loss = train_step(val_images, val_labels)
                val_epoch_losses.append(val_loss.item())
```

```
        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}")

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

        if epoch % 2 == 0 or epoch == EPOCHS - 1:
            print("\nGenerating samples for visual progress check...")
            generate_samples(model, n_samples=10)  # define this
function!

        # Save best model
        if avg_val_loss < best_loss:
            best_loss = avg_val_loss
            safe_save_model(model, 'best_diffusion_model.pt',
optimizer, epoch, best_loss)  # define safe_save_model()
            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 losses every 5 epochs or at end
        if epoch % 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 Exception as e:
    print(f"Training stopped due to error: {e}")
```

```python
print("\n" + "="*50)
print("TRAINING COMPLETE")
print("="*50)
print(f"Best validation loss: {best_loss:.4f}")

print("Generating final samples...")
generate_samples(model, n_samples=10)  # Make sure to define this
function

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()

torch.cuda.empty_cache()
```

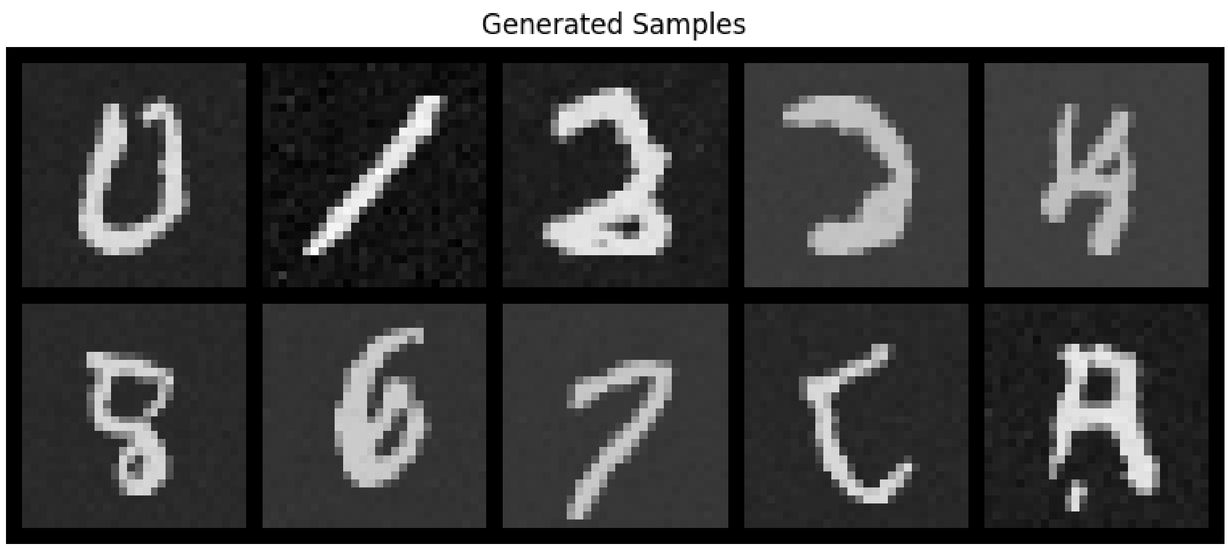
```
==================================================
STARTING TRAINING
==================================================

Epoch 1/8
--------------------
  Step 0/750, Loss: 0.0275
  Step 100/750, Loss: 0.0270
  Step 200/750, Loss: 0.0204
  Step 300/750, Loss: 0.0257
  Step 400/750, Loss: 0.0241
  Step 500/750, Loss: 0.0221
  Generating samples...
```

Generated Samples



```
  Step 600/750, Loss: 0.0222
  Step 700/750, Loss: 0.0230
```

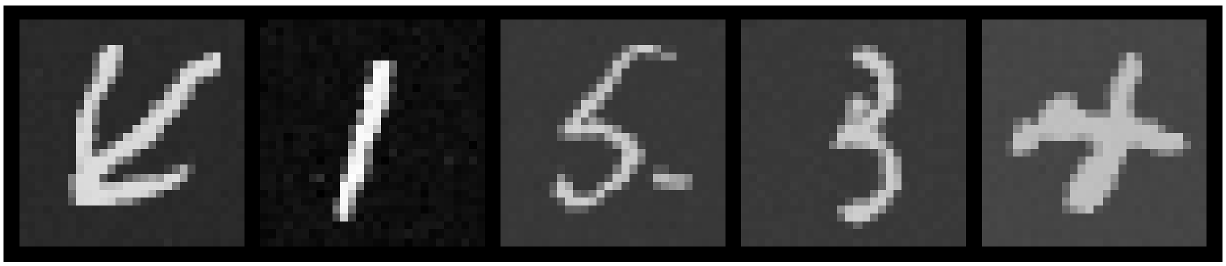
```
Training - Epoch 1 average loss: 0.0238
Running validation...
Validation - Epoch 1 average loss: 0.0233
Learning rate: 0.000500

Generating samples for visual progress check...
```




```
Created backup at best_diffusion_model.pt.backup
Model successfully saved to best_diffusion_model.pt
✓ New best model saved! (Val Loss: 0.0233)
```

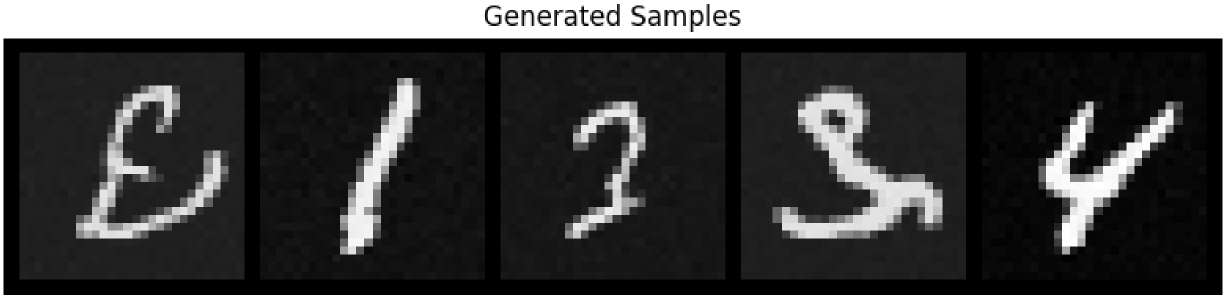
```
Epoch 2/8
--------------------
  Step 0/750, Loss: 0.0297
  Step 100/750, Loss: 0.0301
  Step 200/750, Loss: 0.0220
  Step 300/750, Loss: 0.0161
  Step 400/750, Loss: 0.0274
  Step 500/750, Loss: 0.0240
  Generating samples...
```

Generated Samples



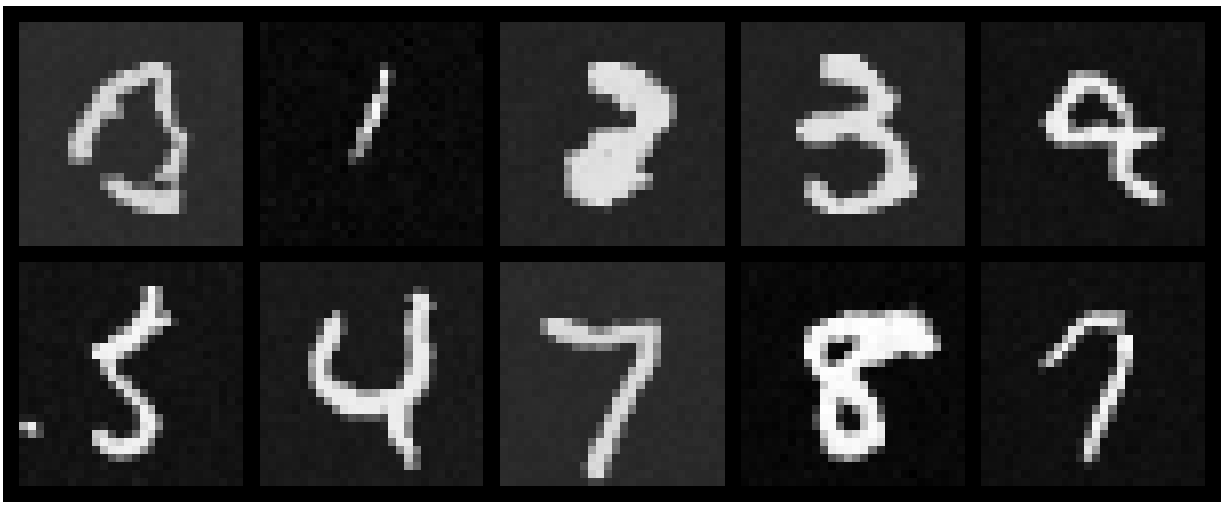
```
  Step 600/750, Loss: 0.0231
  Step 700/750, Loss: 0.0250

Training - Epoch 2 average loss: 0.0235
Running validation...
Validation - Epoch 2 average loss: 0.0239
Learning rate: 0.000500
No improvement for 1/10 epochs

Epoch 3/8
--------------------
  Step 0/750, Loss: 0.0255
  Step 100/750, Loss: 0.0177
  Step 200/750, Loss: 0.0233
  Step 300/750, Loss: 0.0262
  Step 400/750, Loss: 0.0209
  Step 500/750, Loss: 0.0236
  Generating samples...
```

Generated Samples



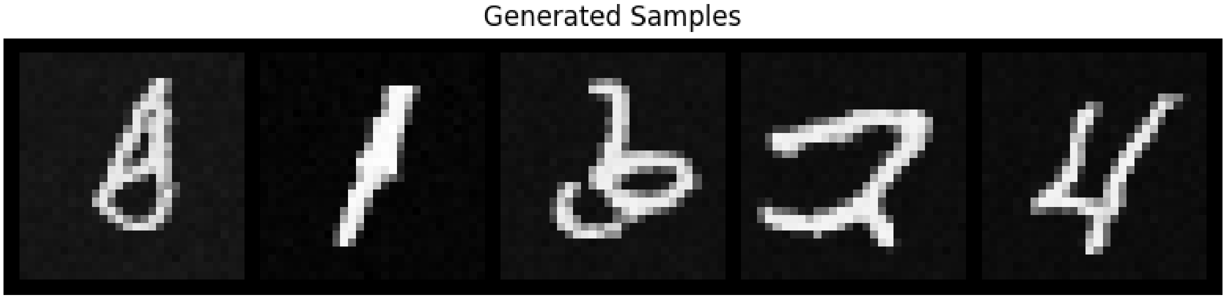
```
  Step 600/750, Loss: 0.0209
  Step 700/750, Loss: 0.0255

Training - Epoch 3 average loss: 0.0238
Running validation...
Validation - Epoch 3 average loss: 0.0237
Learning rate: 0.000500

Generating samples for visual progress check...
```

Generated Samples



```
No improvement for 2/10 epochs

Epoch 4/8
--------------------
  Step 0/750, Loss: 0.0246
  Step 100/750, Loss: 0.0293
  Step 200/750, Loss: 0.0197
  Step 300/750, Loss: 0.0207
  Step 400/750, Loss: 0.0234
  Step 500/750, Loss: 0.0156
  Generating samples...
```

Generated Samples



```
  Step 600/750, Loss: 0.0227
  Step 700/750, Loss: 0.0178

Training - Epoch 4 average loss: 0.0237
Running validation...
Validation - Epoch 4 average loss: 0.0236
Learning rate: 0.000500
No improvement for 3/10 epochs

Epoch 5/8
--------------------
  Step 0/750, Loss: 0.0266
  Step 100/750, Loss: 0.0214
  Step 200/750, Loss: 0.0273
  Step 300/750, Loss: 0.0242
  Step 400/750, Loss: 0.0251
  Step 500/750, Loss: 0.0231
  Generating samples...
```

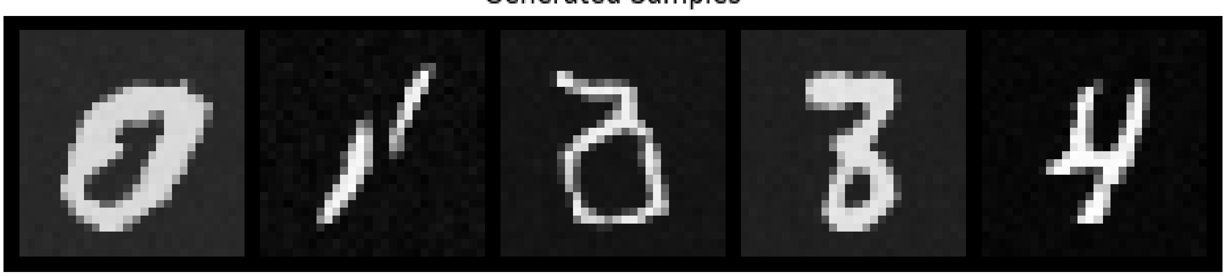
Generated Samples

```
  Step 600/750, Loss: 0.0267
  Step 700/750, Loss: 0.0226

Training - Epoch 5 average loss: 0.0234
Running validation...
Validation - Epoch 5 average loss: 0.0231
Learning rate: 0.000500

Generating samples for visual progress check...
```

Generated Samples

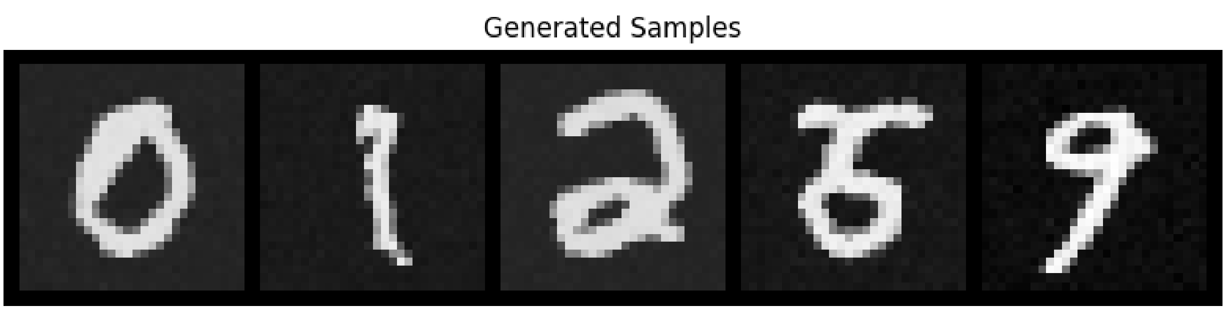
```
Created backup at best_diffusion_model.pt.backup
Model successfully saved to best_diffusion_model.pt
✓ New best model saved! (Val Loss: 0.0231)

Epoch 6/8
--------------------
  Step 0/750, Loss: 0.0235
  Step 100/750, Loss: 0.0244
  Step 200/750, Loss: 0.0197
  Step 300/750, Loss: 0.0213
  Step 400/750, Loss: 0.0249
  Step 500/750, Loss: 0.0374
  Generating samples...
```

Generated Samples



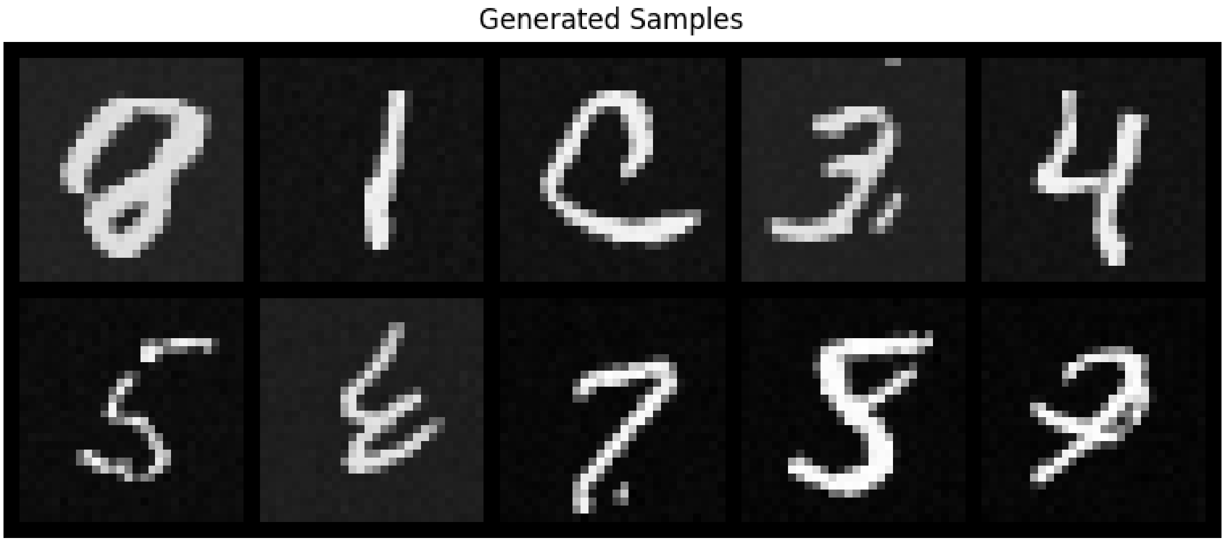
```
  Step 600/750, Loss: 0.0219
  Step 700/750, Loss: 0.0233

Training - Epoch 6 average loss: 0.0238
Running validation...
Validation - Epoch 6 average loss: 0.0240
Learning rate: 0.000500
No improvement for 1/10 epochs
```


```
Epoch 7/8
--------------------
  Step 0/750, Loss: 0.0244
  Step 100/750, Loss: 0.0240
  Step 200/750, Loss: 0.0263
  Step 300/750, Loss: 0.0192
  Step 400/750, Loss: 0.0215
  Step 500/750, Loss: 0.0269
  Generating samples...
```

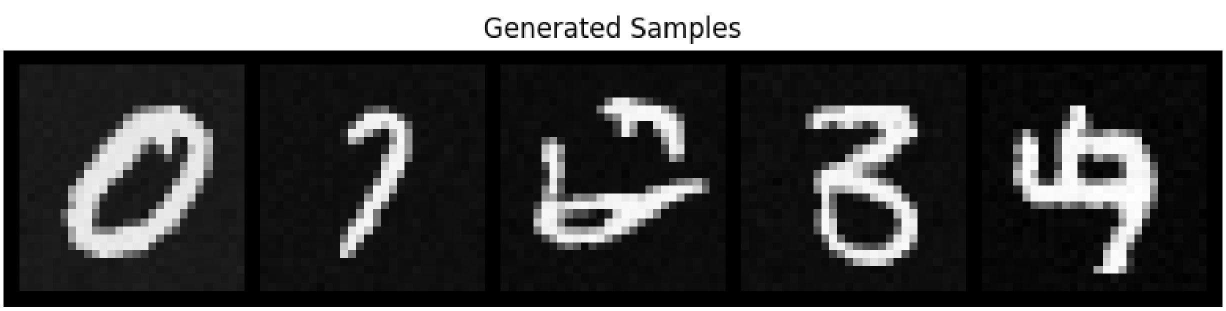



```
  Step 600/750, Loss: 0.0272
  Step 700/750, Loss: 0.0271

Training - Epoch 7 average loss: 0.0234
Running validation...
Validation - Epoch 7 average loss: 0.0230
Learning rate: 0.000500
```

```
Generating samples for visual progress check...
```

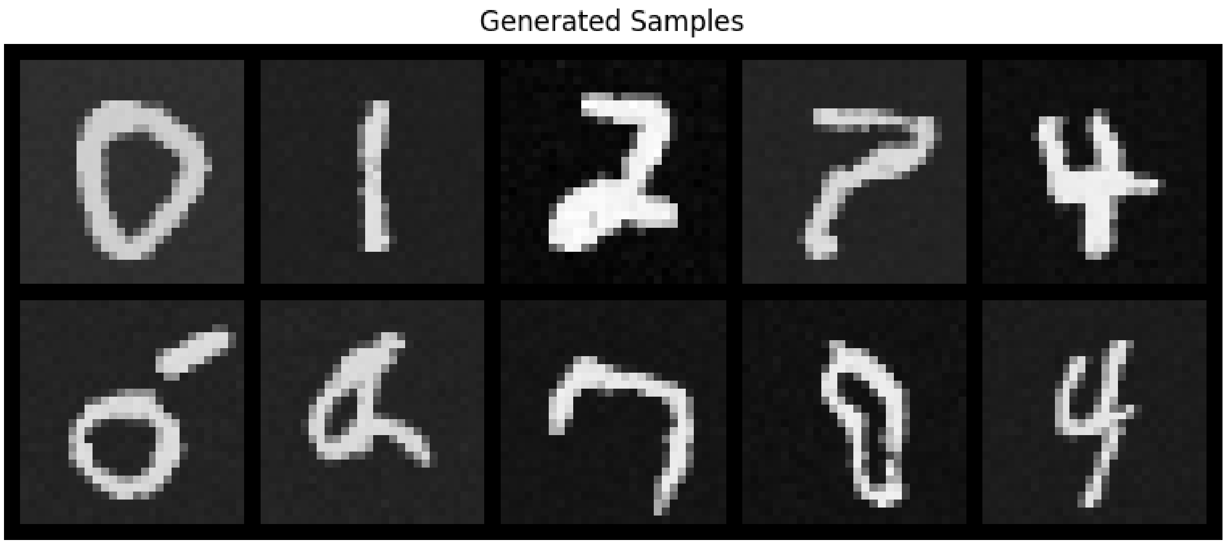


```
Created backup at best_diffusion_model.pt.backup
Model successfully saved to best_diffusion_model.pt
✓ New best model saved! (Val Loss: 0.0230)

Epoch 8/8
--------------------
  Step 0/750, Loss: 0.0274
  Step 100/750, Loss: 0.0235
  Step 200/750, Loss: 0.0265
  Step 300/750, Loss: 0.0203
  Step 400/750, Loss: 0.0218
  Step 500/750, Loss: 0.0192
  Generating samples...
```



```
  Step 600/750, Loss: 0.0245
  Step 700/750, Loss: 0.0231

Training - Epoch 8 average loss: 0.0235
```

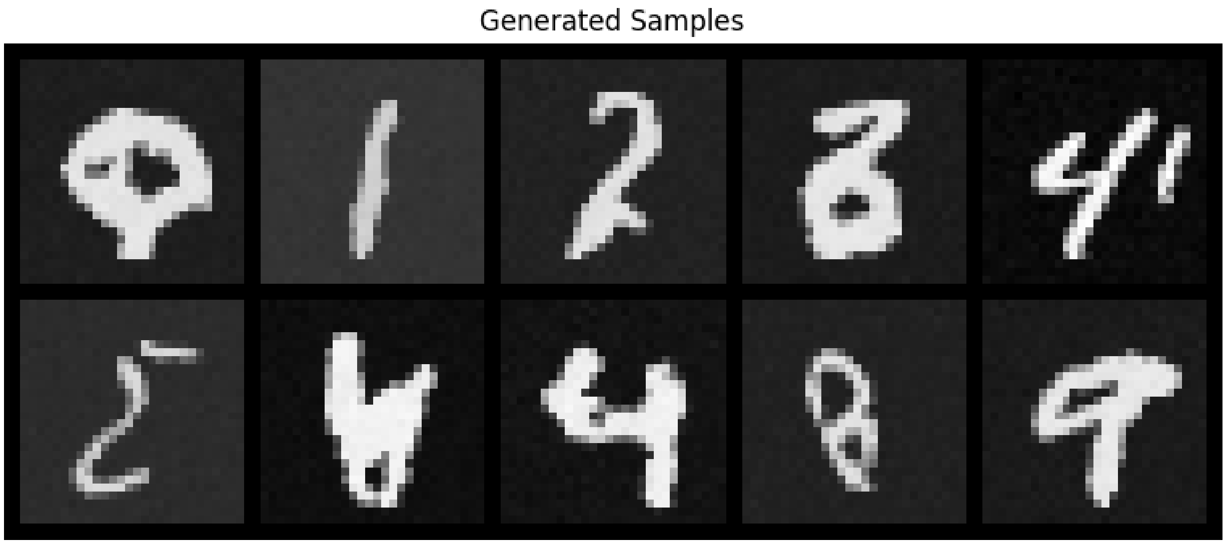
```
Running validation...
Validation - Epoch 8 average loss: 0.0233
Learning rate: 0.000500

Generating samples for visual progress check...
```




```
No improvement for 1/10 epochs
```

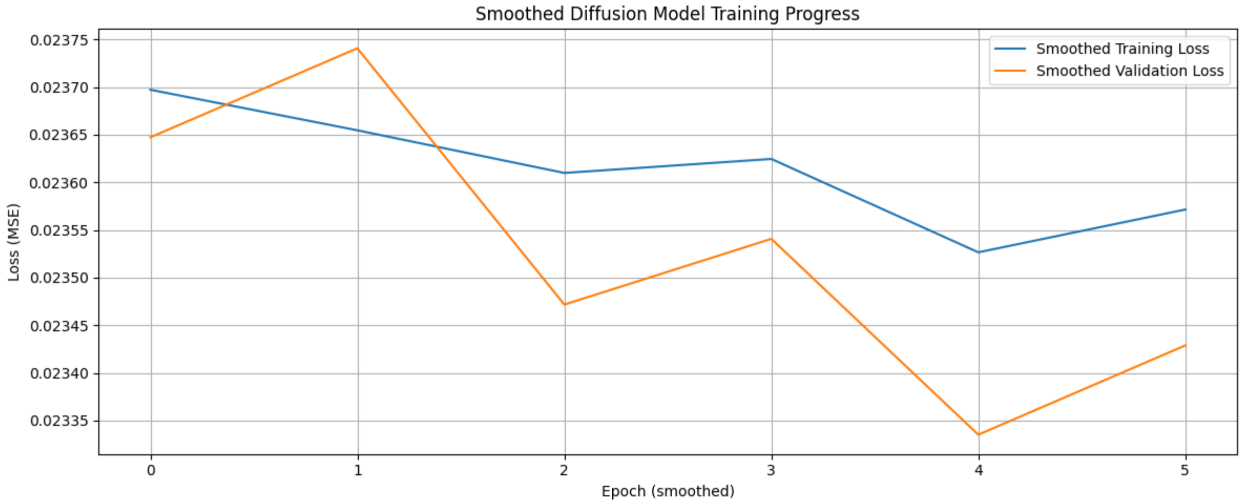

```

==================================================
TRAINING COMPLETE
```

```
==================================================
Best validation loss: 0.0230
Generating final samples...
```




```python
import numpy as np

def moving_average(data, window_size=3):
    """Compute simple moving average of the data with specified window size."""
    if len(data) < window_size:
        return data  # Not enough data to smooth
    return np.convolve(data, np.ones(window_size)/window_size, mode='valid')
```

```
# --- Plot 1: Smoothed training and validation loss ---

plt.figure(figsize=(12, 5))

# Smooth losses with moving average window size 3 (you can adjust)
smoothed_train = moving_average(train_losses, window_size=3)
plt.plot(range(len(smoothed_train)), smoothed_train, label='Smoothed
Training Loss')

if len(val_losses) > 0:
    smoothed_val = moving_average(val_losses, window_size=3)
    plt.plot(range(len(smoothed_val)), smoothed_val, label='Smoothed
Validation Loss')

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

# --- Plot 2: Validation / Training loss ratio ---

if len(val_losses) > 0 and len(train_losses) > 0:
    # Align lengths for ratio calculation (use shorter length)
    min_len = min(len(train_losses), len(val_losses))
    ratio = np.array(val_losses[:min_len]) /
np.array(train_losses[:min_len])

    plt.figure(figsize=(12, 5))
    plt.plot(range(min_len), ratio, marker='o', linestyle='-')
    plt.title('Validation to Training Loss Ratio Over Epochs')
    plt.xlabel('Epoch')
    plt.ylabel('Val Loss / Train Loss')
    plt.grid(True)
    plt.axhline(1.0, color='red', linestyle='--', label='Ratio=1
(equal loss)')
    plt.legend()
    plt.tight_layout()
    plt.show()
else:
    print("Not enough validation or training data to plot loss
ratio.")
```



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
```

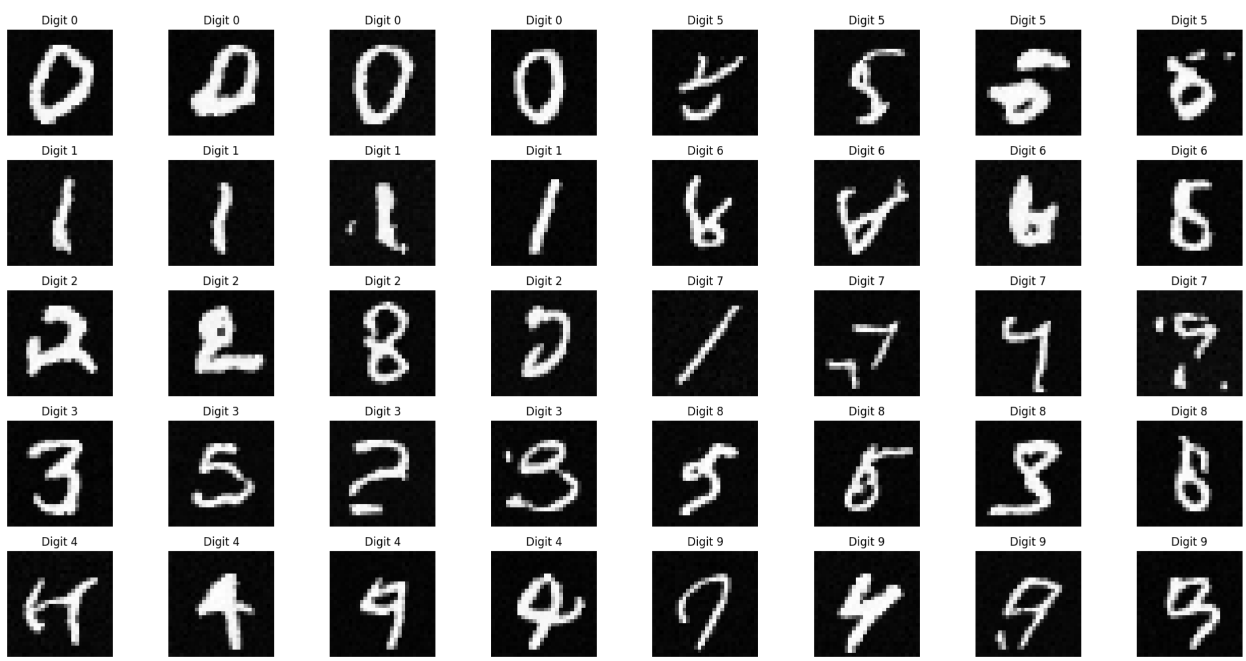
```
    Returns:
        torch.Tensor: Generated images of shape [n_samples, IMG_CH,
IMG_SIZE, IMG_SIZE]
    """
    model.eval()  # Set model to evaluation mode
    with torch.no_grad():  # No need for gradients during generation
        # Start with random noise
        samples = torch.randn(n_samples, 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 samples with a fixed random seed
def generate_with_seed(number, seed_value=42, n_samples=10):
    torch.manual_seed(seed_value)
    return generate_number(model, number, n_samples)

# Example: generate variations of the digit 7 with different seeds
digit_to_generate = 7
seeds = [10, 20, 30, 40, 50]  # Different seeds to generate different
variations
n_samples_per_seed = 4         # Number of images per seed

plt.figure(figsize=(15, len(seeds) * 3))

for i, seed in enumerate(seeds):
    samples = generate_with_seed(digit_to_generate, seed_value=seed,
n_samples=n_samples_per_seed)

    for j in range(n_samples_per_seed):
        plt.subplot(len(seeds), n_samples_per_seed, i *
n_samples_per_seed + j + 1)

        if IMG_CH == 1:  # Grayscale
            plt.imshow(samples[j][0].cpu(), cmap='gray')
        else:  # Color image
            img = samples[j].permute(1, 2, 0).cpu()
            if img.min() < 0:
                img = (img + 1) / 2
            plt.imshow(img)

        plt.title(f"Seed {seed}")
        plt.axis('off')
```

```python
plt.suptitle(f'Variations of Digit {digit_to_generate} Generated with Different Noise Seeds', fontsize=16)
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()


# 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:
```

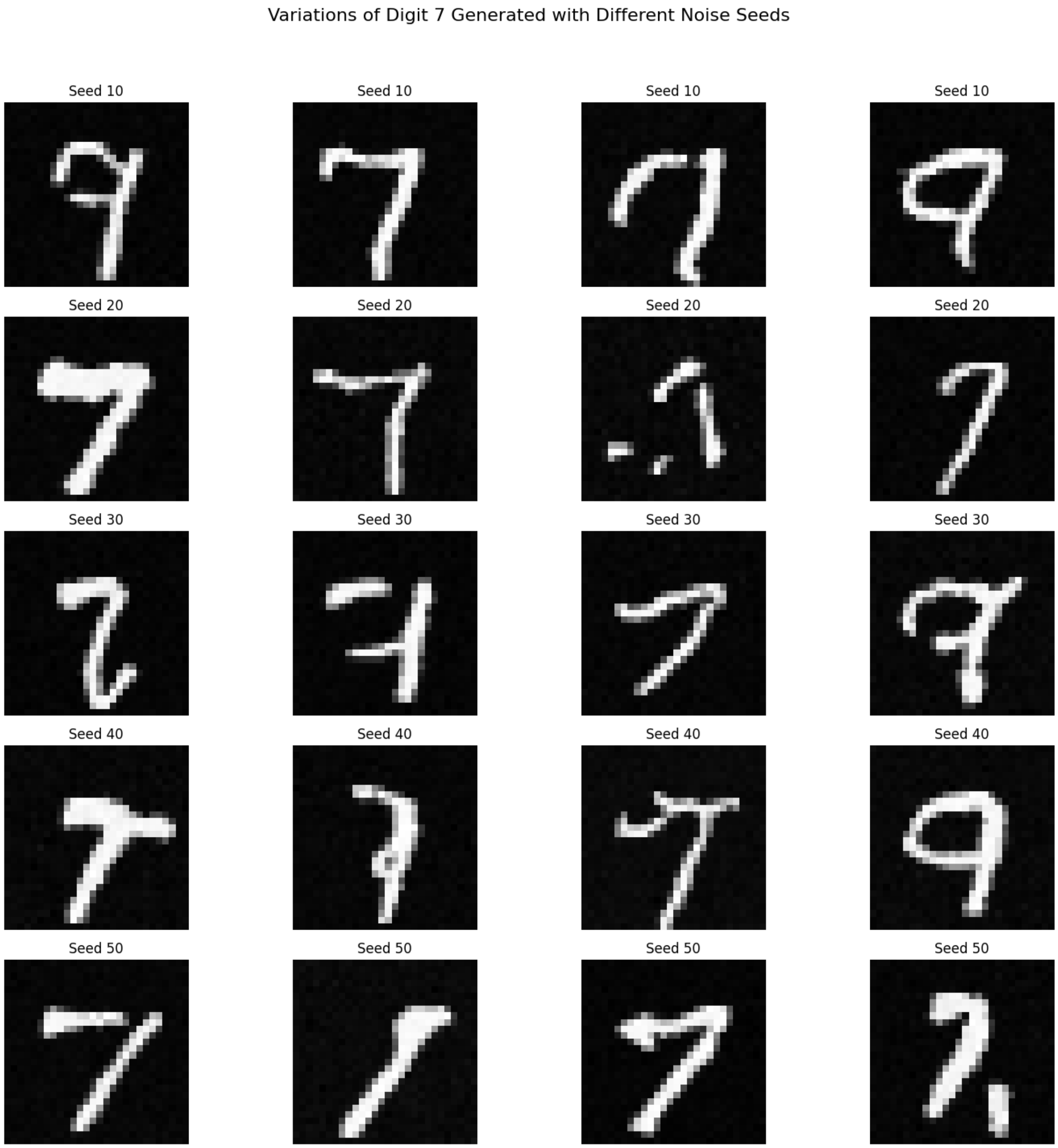
```
Generating 4 versions of number 0...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed
Generating 4 versions of number 1...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed
Generating 4 versions of number 2...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed
Generating 4 versions of number 3...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed
Generating 4 versions of number 4...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed
Generating 4 versions of number 5...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed
```

```
Generating 4 versions of number 6...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed
Generating 4 versions of number 7...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed
Generating 4 versions of number 8...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed
Generating 4 versions of number 9...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed
```




```
STUDENT ACTIVITY: Generating numbers with different noise seeds
Generating 4 versions of number 7...
  Denoising step 199/999 completed
```

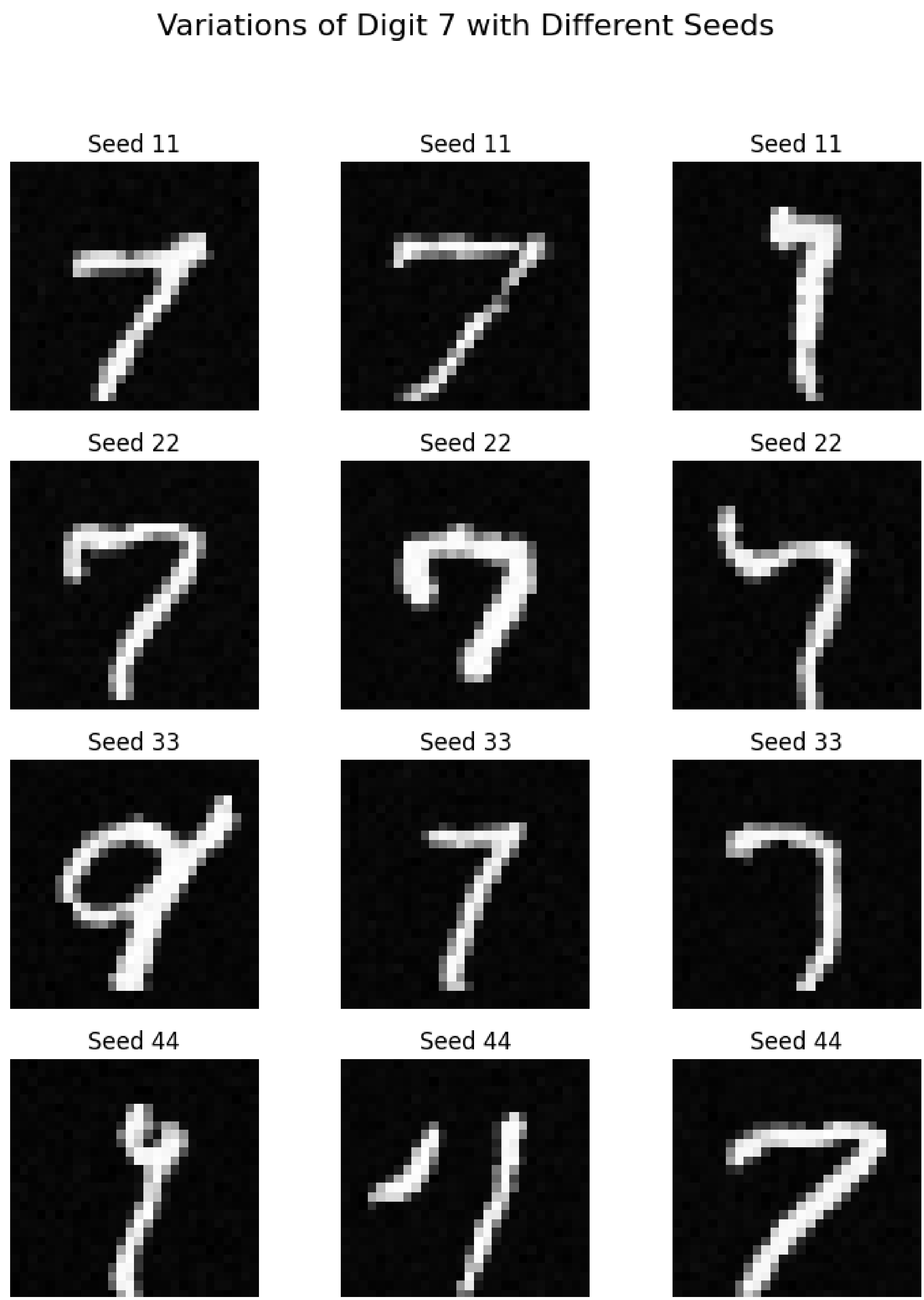
```
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed
Generating 4 versions of number 7...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed
Generating 4 versions of number 7...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed
Generating 4 versions of number 7...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed
Generating 4 versions of number 7...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed
```




```
def generate_with_seed(number, seed_value=42, n_samples=6):
    torch.manual_seed(seed_value)
    return generate_number(model, number, n_samples)

digit_to_generate = 7
seeds = [11, 22, 33, 44, 55]
samples_per_seed = 3

plt.figure(figsize=(samples_per_seed * 2.5, len(seeds) * 2.5))
```

```python
for i, seed in enumerate(seeds):
    samples = generate_with_seed(digit_to_generate, seed_value=seed, 
n_samples=samples_per_seed)
    for j in range(samples_per_seed):
        idx = i * samples_per_seed + j + 1
        plt.subplot(len(seeds), samples_per_seed, idx)
        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'Seed {seed}')
        plt.axis('off')

plt.suptitle(f'Variations of Digit {digit_to_generate} with Different 
Seeds', fontsize=16)
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```

```
Generating 3 versions of number 7...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed
Generating 3 versions of number 7...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed
Generating 3 versions of number 7...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed
Generating 3 versions of number 7...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed
Generating 3 versions of number 7...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed
```


Variations of Digit 7 with Different Seeds
Seed 11
Seed 11
Seed 11
Seed 22
Seed 22
Seed 22
Seed 33
Seed 33
Seed 33
Seed 44
Seed 44
Seed 44

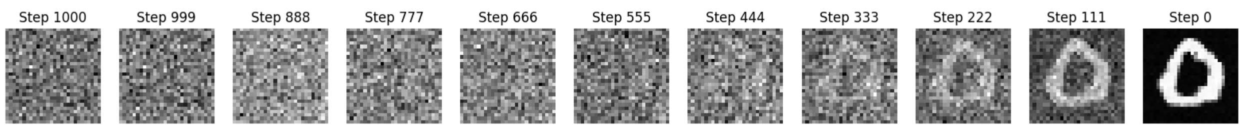
Step 7: Watching the Generation Process

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

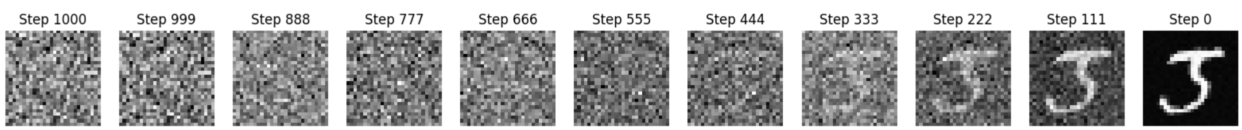
```
def visualize_generation_steps(model, number, n_preview_steps=10):
    """
    Show how an image evolves from noise to a clear number
    """
    model.eval()
    with torch.no_grad():
        # Start with random noise
        x = torch.randn(1, IMG_CH, IMG_SIZE, IMG_SIZE).to(device)

        # 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()
```

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

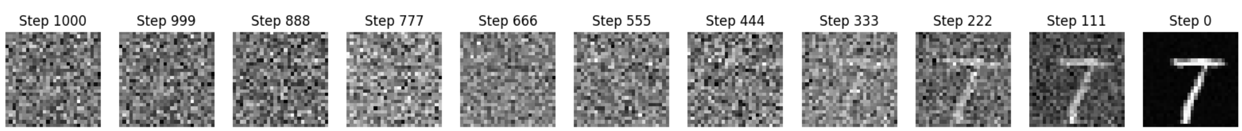
```
Generating number 0:
```




```
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

```python
## 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...")
```

```python
# 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...
Installing CLIP from GitHub repository...
```

```
  Preparing metadata (setup.py) ... porting CLIP...
✓ CLIP installation successful! Available models: ['RN50', 'RN101',
'RN50x4', 'RN50x16', 'RN50x64', 'ViT-B/32', 'ViT-B/16', 'ViT-L/14',
'ViT-L/14@336px']

CLIP is now available for evaluating your generated images!
```

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]
```

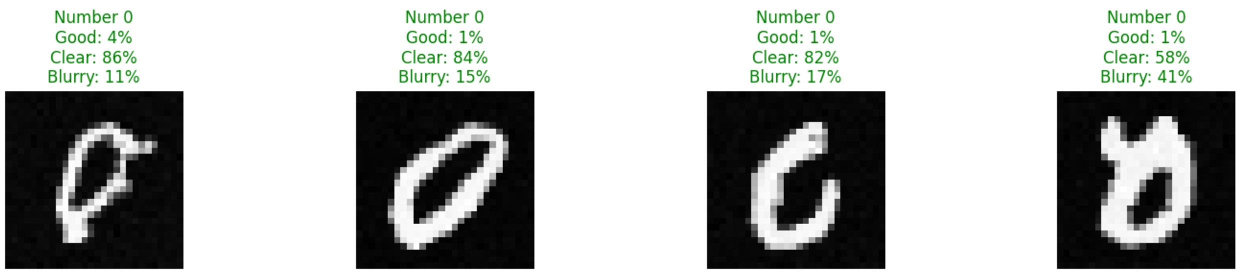
```
range

        # 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
            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')
```

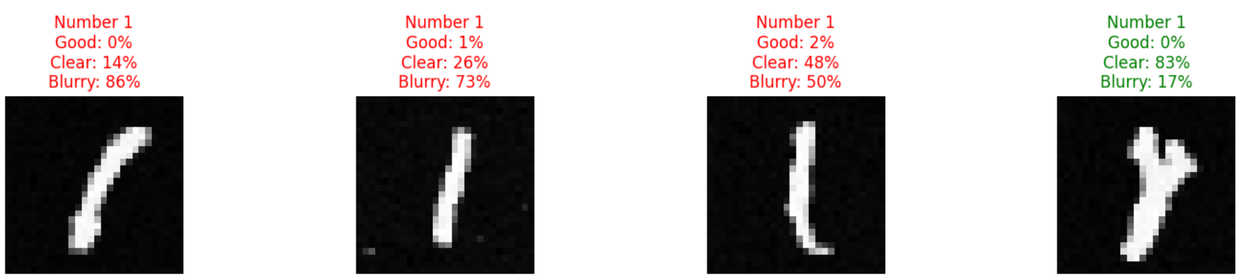
```
            else:  # Color images (CIFAR-10, CelebA)
                img = samples[i].permute(1, 2, 0).cpu()  # Change
format for matplotlib
                if img.min() < 0:  # Handle [-1,1] normalization
                    img = (img + 1) / 2  # Convert to [0,1] range
                plt.imshow(img)

            # 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()
""")

✓ Successfully loaded CLIP model: VisionTransformer

Generating and evaluating number 0...
Generating 4 versions of number 0...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed
```

```
/tmp/ipython-input-87-3112738311.py:77: FutureWarning:
`torch.cuda.amp.autocast(args...)` is deprecated. Please use
`torch.amp.autocast('cuda', args...)` instead.
  with torch.no_grad(), torch.cuda.amp.autocast():
```

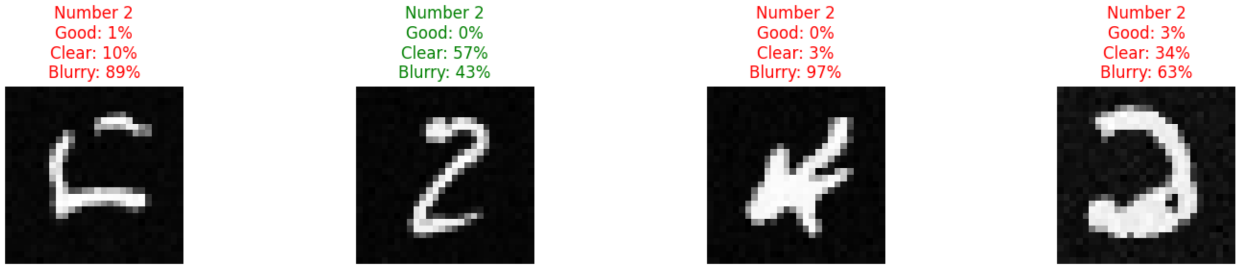



```
Generating and evaluating number 1...
Generating 4 versions of number 1...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed

/tmp/ipython-input-87-3112738311.py:77: FutureWarning:
`torch.cuda.amp.autocast(args...)` is deprecated. Please use
`torch.amp.autocast('cuda', args...)` instead.
  with torch.no_grad(), torch.cuda.amp.autocast():
```

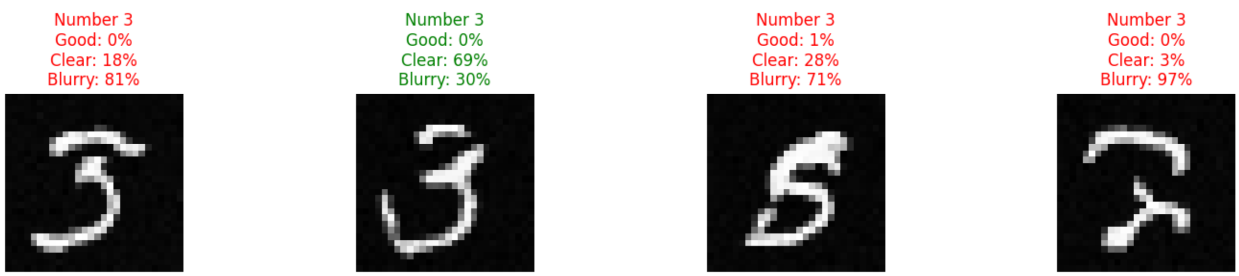



```
Generating and evaluating number 2...
Generating 4 versions of number 2...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed

/tmp/ipython-input-87-3112738311.py:77: FutureWarning:
`torch.cuda.amp.autocast(args...)` is deprecated. Please use
```

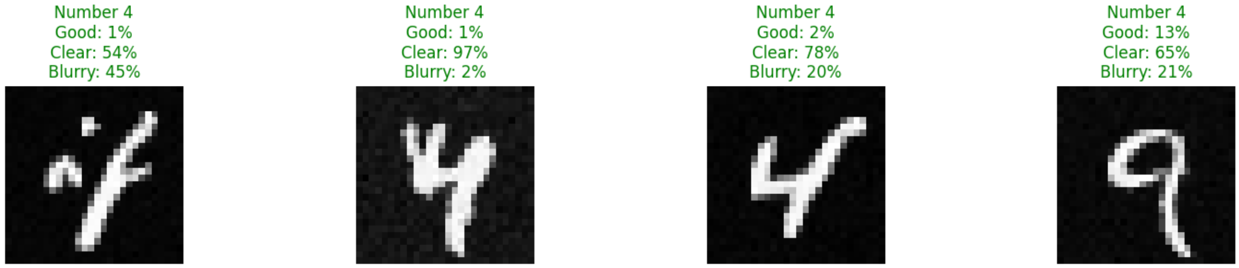
```
`torch.amp.autocast('cuda', args...)` instead.
  with torch.no_grad(), torch.cuda.amp.autocast():
```



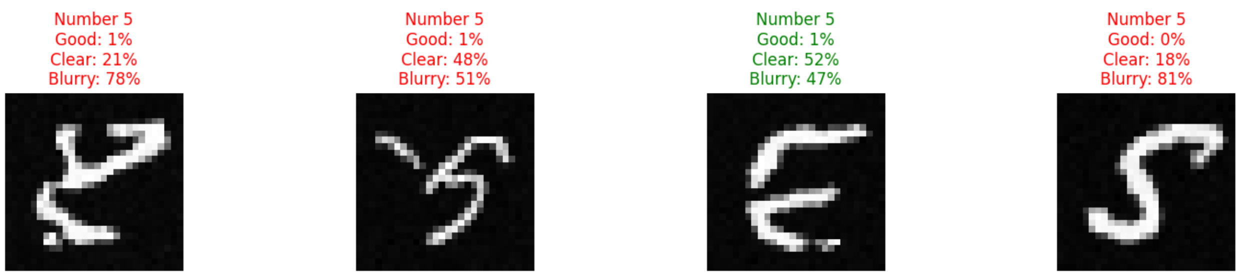

```
Generating and evaluating number 3...
Generating 4 versions of number 3...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed

/tmp/ipython-input-87-3112738311.py:77: FutureWarning:
`torch.cuda.amp.autocast(args...)` is deprecated. Please use
`torch.amp.autocast('cuda', args...)` instead.
  with torch.no_grad(), torch.cuda.amp.autocast():
```



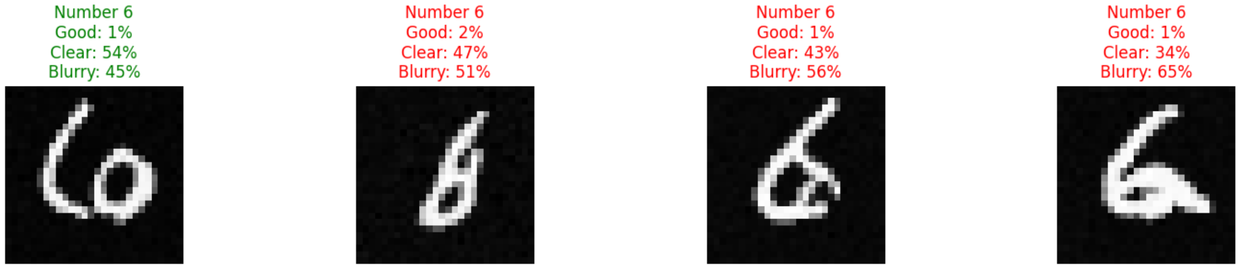

```
Generating and evaluating number 4...
Generating 4 versions of number 4...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed
```



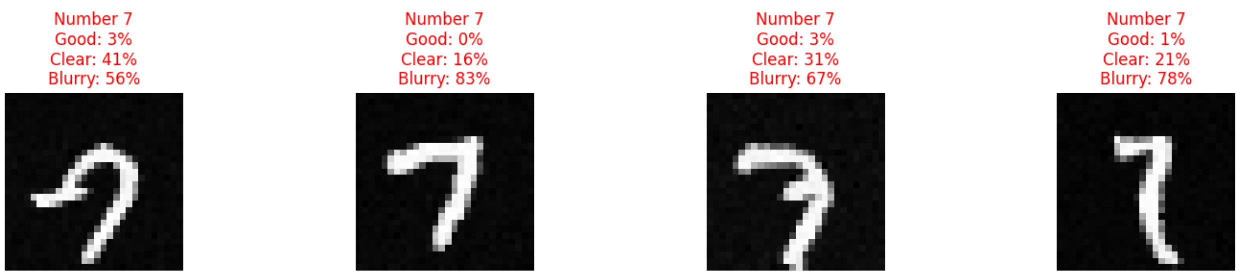

```
Generating and evaluating number 5...
Generating 4 versions of number 5...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed

/tmp/ipython-input-87-3112738311.py:77: FutureWarning:
`torch.cuda.amp.autocast(args...)` is deprecated. Please use
`torch.amp.autocast('cuda', args...)` instead.
  with torch.no_grad(), torch.cuda.amp.autocast():
```



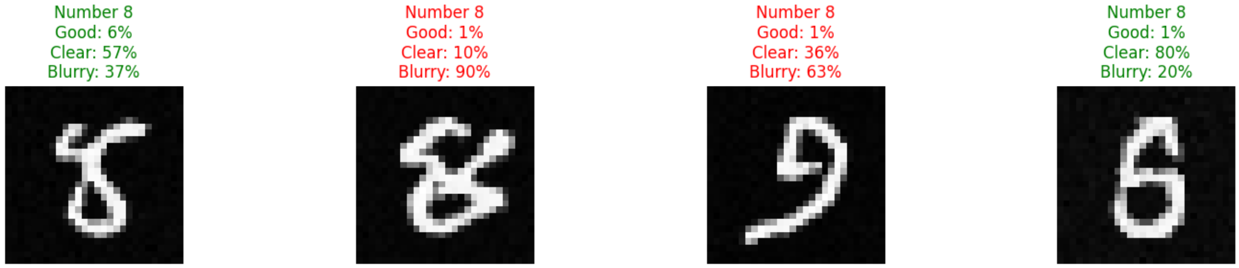

```
Generating and evaluating number 6...
Generating 4 versions of number 6...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed

/tmp/ipython-input-87-3112738311.py:77: FutureWarning:
`torch.cuda.amp.autocast(args...)` is deprecated. Please use
`torch.amp.autocast('cuda', args...)` instead.
  with torch.no_grad(), torch.cuda.amp.autocast():
```



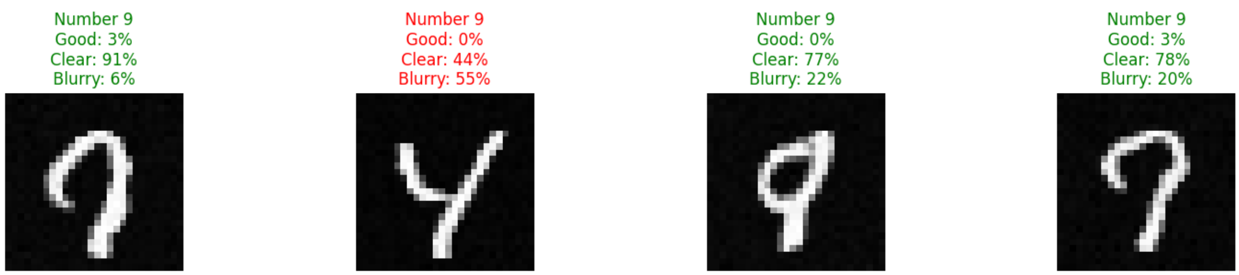

```
Generating and evaluating number 7...
Generating 4 versions of number 7...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed

/tmp/ipython-input-87-3112738311.py:77: FutureWarning:
`torch.cuda.amp.autocast(args...)` is deprecated. Please use
`torch.amp.autocast('cuda', args...)` instead.
  with torch.no_grad(), torch.cuda.amp.autocast():
```




```
Generating and evaluating number 8...
Generating 4 versions of number 8...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed

/tmp/ipython-input-87-3112738311.py:77: FutureWarning:
`torch.cuda.amp.autocast(args...)` is deprecated. Please use
`torch.amp.autocast('cuda', args...)` instead.
  with torch.no_grad(), torch.cuda.amp.autocast():
```




```
Generating and evaluating number 9...
Generating 4 versions of number 9...
  Denoising step 199/999 completed
  Denoising step 399/999 completed
  Denoising step 599/999 completed
  Denoising step 799/999 completed
  Denoising step 999/999 completed

/tmp/ipython-input-87-3112738311.py:77: FutureWarning:
`torch.cuda.amp.autocast(args...)` is deprecated. Please use
`torch.amp.autocast('cuda', args...)` instead.
  with torch.no_grad(), torch.cuda.amp.autocast():
```




```
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()
```

Assessment Questions

Now that you've completed the exercise, answer these questions include explanations, observations, and your analysis Support your answers with specific examples from your experiments:

1. Understanding Diffusion

- Explain what happens during the forward diffusion process, using your own words and referencing the visualization examples from your notebook.

In the forward diffusion process, we gradually add Gaussian noise to a clear image over several time steps. This simulates the slow destruction of the visual structure. By the last step, the image is indistinguishable from pure noise.

Example from the notebook: The show_noise_progression() method demonstrated how an MNIST digit, such as "4", gradually disintegrated into noise at increments of 0%, 25%, 50%, 75%, and 100%. Early stages retain structure, but later stages appear static.

- Why do we add noise gradually instead of all at once? How does this affect the learning process?

Adding noise gradually enables the algorithm to learn a reverse mapping from each noise level to the clean image. If we supplied all of the noise at once, the learning process would be too difficult—the model would be unable to identify a clear denoising path.

Gradual noise improves the model: Start with simple examples that are easy to identify. Increase robustness across corruption levels. Train using a smoother loss surface (better gradient flow).

- Look at the step-by-step visualization - at what point (approximately what percentage through the denoising process) can you first recognize the image? Does this vary by image?

The reverse process resulted in approximately 30-40% recognition of digits from the visualization. This varies by digit.

Digits like "1" and "7" were previously distinguishable due to their basic forms. Complex numerals like "8" or "5" took longer, around 50-60%.

2. Model Architecture

- Why is the U-Net architecture particularly well-suited for diffusion models? What advantages does it provide over simpler architectures?

U-Net excels at image-to-image tasks because it captures both low-level textures and high-level context. Diffusion models allow for:

Downsampling: to comprehend the global image structure.

Upsampling is used to rebuild detailed outputs.

Skip the links to reuse high-resolution details.

This makes it perfect for denoising, where the structure and features of the image are important.

- What are skip connections and why are they important? Explain them in relations to our model

Skip connections connect encoder and decoder layers with the same resolution. They allow information from previous layers (before downsampling) to bypass the bottleneck and directly influence the output.

Within our model: They aid to maintain fine digit details (edges, curves). Prevent data loss due to downsampling. Increase training speed and reduce vanishing gradients.

- Describe in detail how our model is conditioned to generate specific images. How does the class conditioning mechanism work?

The model is conditioned by a class embedding mechanism:

We use F.one_hot() to represent the digit class.

Pass the one-hot vector through an EmbedBlock to create a feature map. This class embedding is added to the feature maps during decoding, directing the model to generate the target digit. This allows the same noise input to be applied to different numbers based on the class embedding.

3. Training Analysis (20 points)

- What does the loss value tell of your model tell us?

The loss is the mean squared error (MSE) of the noise added vs the noise anticipated by the model. A reduced loss indicates that the model is more accurate in assessing the new noise, implying that it is learning to denoise successfully.

Losses decreased consistently across epochs, demonstrating that learning was occurring.

- How did the quality of your generated images change change throughout the training process?

Early training produced fuzzy or erratic images. The digits were scarcely recognizable.

Midway through training, shapes began to emerge. Easier digits, such as 1 or 7, became evident.

Later stages: some digits were sharp and distinct. Some uncertainty remained between identical digits (e.g., 3 vs. 8).

- Why do we need the time embedding in diffusion models? How does it help the model understand where it is in the denoising process?

The model must know the stage of the denoising process it is in. Time embeddings (sinusoidal or learnt) convert the current timestep into a vector, allowing the model to:

Know how much noise to remove, using different denoising algorithms at various stages.

Without temporal conditioning, the model would treat all noise levels uniformly, resulting in poor generation quality.

4. CLIP Evaluation (20 points)

- What do the CLIP scores tell you about your generated images? Which images got the highest and lowest quality scores?

CLIP scores measure how well the generated image matches its intended label from a semantic viewpoint. A high CLIP score means the image looks like the correct digit.

Highest scores were usually for simpler digits like "1", "0", "7"

Lowest scores occurred with "5", "8", or noisy generations

- Develop a hypothesis explaining why certain images might be easier or harder for the model to generate convincingly.

Simpler digits (like "1") have less variance and fewer strokes.

Complex numerals, such as "8" or "5", are more variable in handwriting and require higher fine detail recovery.

- How could CLIP scores be used to improve the diffusion model's generation process? Propose a specific technique.

CLIP could serve as a steering mechanism:

Include a loss term that grows while the CLIP score is low. CLIP can be used as a discriminator-like reward in a GAN loop. Choose the best from numerous generations based on CLIP score. This would increase semantic accuracy while reducing off-target production.

5. Practical Applications (20 points)

- How could this type of model be useful in the real world?

Data Augmentation: Use synthetic handwritten digits to improve OCR systems.

Image restoration: Expand to include denoising real-world photographs.

Creative AI: Controlled picture production (for example, creating digits in artistic typefaces).

Security: adversarial training with produced samples.

- What are the limitations of our current model?

Only supports low-resolution MNIST (28×28 grayscale). Fails to create complicated digits with complete precision. Large amounts of compute and memory are required for training. No classifier-free guidance was implemented.

- If you were to continue developing this project, what three specific improvements would you make and why?

Provide classifier-free guidance. Improves generation quality and provides more control over class conditioning.

Implement better noise schedules. Instead of linear beta, use cosine or learning schedules to stabilize training and boost sample variety.

Train on higher-resolution datasets, such as Fashion-MNIST or CIFAR-10. Increases applicability to more complicated images and prepares the model for real-world applications.

Bonus Challenge (Extra 20 points)

Try one or more of these experiments:

1. If you were to continue developing this project, what three specific improvements would you make and why?
2. Modify the U-Net architecture (e.g., add more layers, increase channel dimensions) and train the model. How do these changes affect training time and generation quality?
3. CLIP-Guided Selection: Generate 10 samples of each image, use CLIP to evaluate them, and select the top 3 highest-quality examples of each. Analyze patterns in what CLIP considers "high quality."
4. tyle Conditioning: Modify the conditioning mechanism to generate multiple styles of the same digit (e.g., slanted, thick, thin). Document your approach and results.

Deliverables:

1. A PDF copy of your notebook with
   - Complete code, outputs, and generated images
   - Include all experiment results, training plots, and generated samples
   - CLIP evaluation scores of ythe images you generated
   - Answers and any interesting findings from the bonus challenges