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

Introduction

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

What We'll Build

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

Dataset Options

This lab offers flexibility based on your available computational resources:

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

Resource Requirements

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

Before You Start

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

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

Step 1: Setting Up Our Tools

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

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

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

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

# --- Tensor manipulation ---
import random  # For random operations
from einops.layers.torch import Rearrange  # For reshaping tensors in neural networks
```

```python
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/python3.12/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

```python
# 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

import torchvision # Added import
import torchvision.transforms as transforms
from torch.utils.data import DataLoader # Added import
import os # Added import

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
```

```python

# Enter your code here:
# --- Load MNIST dataset ---
train_dataset = torchvision.datasets.MNIST(
    root="./data",
    train=True,
    transform=transform,
    download=True
)

test_dataset = torchvision.datasets.MNIST(
    root="./data",
    train=False,
    transform=transform,
    download=True
)

# --- Create DataLoaders ---
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=os.cpu_count())
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=os.cpu_count())

print(f"✅ MNIST dataset loaded successfully! Total training samples: {len(train_dataset)}")
print(f"Example image shape: {train_dataset[0][0].shape}")
```

```
✅ MNIST dataset loaded successfully! Total training samples: 60000
Example image shape: torch.Size([1, 28, 28])
```

```python
#==============================================================================
# VALIDATING DATASET SELECTION & GPU MEMORY (MNIST ONLY)
#==============================================================================

# --- Helper: probe presence of common dataset vars ---
has_explicit_flag = 'dataset' in globals()
has_train_dataset = 'train_dataset' in globals()
has_test_dataset  = 'test_dataset' in globals()
has_train_loader  = 'train_loader' in globals()
has_test_loader   = 'test_loader' in globals()

if not (has_explicit_flag or has_train_dataset or has_test_dataset or has_train_loader or has_test_loader):
    raise ValueError("""
❌ ERROR: No dataset selected! Please uncomment the MNIST dataset option.
Available option:
1. MNIST (Basic) - 2GB GPU
""")

# --- Infer dataset name if not explicitly set ---
def _infer_dataset_name():
    if 'dataset' in globals() and isinstance(dataset, str):
        return dataset
    if 'train_dataset' in globals():
        return train_dataset.__class__.__name__
    if 'train_loader' in globals():
        try:
            return train_loader.dataset.__class__.__name__
        except Exception:
            pass
    if 'IMG_SIZE' in globals() and 'IMG_CH' in globals():
        if IMG_SIZE == 28 and IMG_CH == 1:
            return "MNIST"
    return "Unknown"

dataset_name = _infer_dataset_name()

if dataset_name == "MNIST" and 'dataset' not in globals():
    dataset = "MNIST"

print(f"📦 Detected dataset: {dataset_name}")

# --- GPU memory requirement (MNIST only) ---
required_gb = 2

if torch.cuda.is_available():
    props = torch.cuda.get_device_properties(0)
    gpu_gb = props.total_memory / 1e9
    print(f"💻 GPU: {props.name}")
    print(f"🧠 VRAM: {gpu_gb:.2f} GB (recommended ≥ {required_gb} GB for MNIST)")
    if gpu_gb < required_gb:
        print("⚠️ VRAM may be insufficient — consider reducing batch size.")
    else:
        print("✅ GPU memory check passed for MNIST.")
else:
    print("⚠️ No GPU detected. Training will be slow on CPU. Enable GPU in Colab if possible.")

```

```
📦 Detected dataset: MNIST
💻 GPU: Tesla T4
🧠 VRAM: 15.83 GB (recommended ≥ 2 GB for MNIST)
✅ GPU memory check passed for MNIST.
```

```
# --- Sample batch properties ---
_probe_loader = DataLoader(train_dataset, batch_size=16, shuffle=True, num_workers=0)
xb, yb = next(iter(_probe_loader))  # xb: [B, C, H, W], yb: [B]

print("Batch shape:", tuple(xb.shape))
print("DType:", xb.dtype, "| Labels dtype:", yb.dtype)
print("Image size (C,H,W):", tuple(xb.shape[1:]))
print("Value range (min,max):", float(xb.min()), float(xb.max()))
print("Mean/std (first batch):", xb.mean().item(), xb.std().item())
print("Unique labels in batch:", torch.unique(yb).tolist())

# Optional: quick visual sanity check (comment out if running headless)
# grid = make_grid(xb[:16], nrow=8, normalize=True, value_range=(-1, 1), padding=1)
# plt.figure(figsize=(6,3))
# plt.imshow(grid.permute(1, 2, 0).cpu().numpy(), cmap="gray")
# plt.axis("off")
# plt.title("MNIST sample batch")
# plt.show()

```

```
Batch shape: (16, 1, 28, 28)
DType: torch.float32 | Labels dtype: torch.int64
Image size (C,H,W): (1, 28, 28)
Value range (min,max): -1.0 1.0
Mean/std (first batch): -0.7353366017341614 0.6148757338523865
Unique labels in batch: [0, 1, 2, 3, 4, 5, 6, 8, 9]
```

Step 3: Building Our Model Components

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

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

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

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

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

        # --- Layers: Conv → GroupNorm → GELU ---
        self.block = nn.Sequential(
```

```python
            nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=1),  # preserves spatial size
            nn.GroupNorm(num_groups=group_size, num_channels=out_ch),
            nn.GELU()
        )

    def forward(self, x):
        # Pass input through the sequential block
        return self.block(x)

```

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

        Args:
            in_chs (int): Number of input channels
            group_size (int): Number of groups for GroupNorm
        """
        super().__init__()
        # Space-to-depth: (B, C, H, W) -> (B, 4C, H/2, W/2)
        self.rearrange = Rearrange('b c (h s1) (w s2) -> b (c s1 s2) h w', s1=2, s2=2)

        # After rearrange, channels become 4 * in_chs.
        # Typical U-Net encoder doubles channels at each downsample:
        out_chs = in_chs * 2
        # Fix: The input channels to GELUConvBlock should match the output channels of rearrange (in_chs * 4)
        self.conv = GELUConvBlock(in_chs * 4, out_chs, group_size)

    def forward(self, x):
        x = self.rearrange(x)   # downsample H,W by 2; multiply channels by 4
        x = self.conv(x)        # process features
        return x
```

```python
#Let's implement the upsampling block for our U-Net architecture:
import torch.nn as nn # Import nn module
import torch.nn.functional as F # Import F module

class DownBlock(nn.Module):
    """
    Downsampling block for encoding path in U-Net architecture.

    This block:
    1. Processes input features with two convolutional blocks to change channels
    2. Downsamples spatial dimensions by 2x using a strided convolution

    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__()

        # --- Layers: Conv → GN → GELU → Conv → GN → GELU → Downsample ---
        self.model = nn.Sequential(
            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
            # Spatial downsampling using a strided convolution
            nn.Conv2d(out_chs, out_chs, kernel_size=3, stride=2, padding=1)
        )

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

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

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

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

```
# Now let's implement the upsampling block for our U-Net architecture:
class UpBlock(nn.Module):
    """
    Upsampling block for decoding path in U-Net architecture.

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

    Args:
        in_chs (int): Number of input channels from the previous layer (decoder)
        out_chs (int): Number of output channels for this stage
        group_size (int): Number of groups for GroupNorm
    """
    def __init__(self, in_chs, out_chs, group_size):
        super().__init__()

        # 2× upsampling via transposed convolution: keeps channels at in_chs
        self.up = nn.ConvTranspose2d(in_chs, in_chs, kernel_size=2, stride=2)

        # After upsample, we concatenate with skip (assume skip has in_chs channels in a symmetric U-Net)
        fused_chs = in_chs * 2

        # Two Conv→GN→GELU blocks to refine fused features
        self.refine = nn.Sequential(
            GELUConvBlock(fused_chs, out_chs, group_size),
            GELUConvBlock(out_chs, out_chs, group_size)
        )

        # Log the configuration for debugging
        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): Decoder input [B, in_chs, H, W]
            skip (torch.Tensor): Encoder skip [B, in_chs, 2H, 2W]

        Returns:
            torch.Tensor: [B, out_chs, 2H, 2W]
        """
        # 1) Upsample decoder features to match skip spatial size
        x = self.up(x)  # -> [B, in_chs, 2H, 2W]

        # Handle potential off-by-one mismatches (odd sizes) with pad/crop
        if x.shape[-2:] != skip.shape[-2:]:
            dh = skip.shape[-2] - x.shape[-2]
            dw = skip.shape[-1] - x.shape[-1]
            # pad (right/bottom) if smaller
            x = F.pad(x, (0, max(0, dw), 0, max(0, dh)))
            # crop if we overshot
            x = x[..., :skip.shape[-2], :skip.shape[-1]]

        # 2) Concatenate along channels
        x = torch.cat([x, skip], dim=1)  # [B, in_chs*2, 2H, 2W]

        # 3) Refine fused features
        return self.refine(x)

```

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

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

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

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

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

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


```

```
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

        # --- Class embedding model ---
        self.model = nn.Sequential(
            nn.Linear(input_dim, emb_dim),
            nn.GELU(),
            nn.Linear(emb_dim, emb_dim),
            nn.GELU(),
            nn.Unflatten(1, (emb_dim, 1, 1))  # reshape for broadcasting with feature maps
        )

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

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

        Returns:
            torch.Tensor: Class embeddings [batch_size, emb_dim, 1, 1]
        """
        # If input is class indices, convert to one-hot
        if x.ndim == 1 or x.shape[1] == 1:
            x = F.one_hot(x.long().squeeze(), num_classes=self.input_dim).float()

        x = x.view(-1, self.input_dim)
        return self.model(x)

```

```
# Main U-Net model that puts everything together
class UNet(nn.Module):
    """
    U-Net architecture for diffusion models with time and class conditioning.
    """
    def __init__(self, T, img_ch, img_size, down_chs, t_embed_dim, c_embed_dim):
        super().__init__()
        self.T = T
        self.img_ch = img_ch
        self.img_size = img_size
        self.down_chs = down_chs
        self.group_size = 8  # default GN group size (GELUConvBlock will auto-fix if not divisible)

        base_ch = down_chs[0]
        deepest_ch = down_chs[-1]

        # ---- Time embedding: sinusoidal -> MLP -> [B, deepest_ch, 1, 1]
        self.time_mlp = nn.Sequential(
            SinusoidalPositionEmbedBlock(t_embed_dim),
            nn.Linear(t_embed_dim, t_embed_dim * 4),
            nn.GELU(),
            nn.Linear(t_embed_dim * 4, deepest_ch),
            nn.Unflatten(1, (deepest_ch, 1, 1)),
        )

        # ---- Class embedding: one-hot / index -> MLP -> project to deepest_ch -> [B, deepest_ch, 1, 1]
        # Uses global N_CLASSES defined earlier
        self.class_embed = EmbedBlock(input_dim=N_CLASSES, emb_dim=c_embed_dim)
        self.class_to_ch = nn.Conv2d(c_embed_dim, deepest_ch, kernel_size=1)

        # ---- Initial feature extraction
        self.in_conv = GELUConvBlock(img_ch, base_ch, self.group_size)

        # ---- Down path: list of DownBlocks
        downs = []
        for i in range(len(down_chs) - 1):
            downs.append(DownBlock(down_chs[i], down_chs[i + 1], self.group_size))
        self.down_blocks = nn.ModuleList(downs)

        # ---- Mid (bottleneck) blocks
        self.mid1 = GELUConvBlock(deepest_ch, deepest_ch, self.group_size)
        self.mid2 = GELUConvBlock(deepest_ch, deepest_ch, self.group_size)

        # ---- Up path: list of UpBlocks (reverse order)
        ups = []
        for i in reversed(range(len(down_chs) - 1)):
            # Decoder input channels at this stage = channels from deeper level
            dec_in = down_chs[i + 1]
            dec_out = down_chs[i]
            ups.append(UpBlock(in_chs=dec_in, out_chs=dec_out, group_size=self.group_size))
        self.up_blocks = nn.ModuleList(ups)

        # To make concatenation work even if skip channels != decoder in_chs,
        # create 1×1 projections for each skip (align to decoder in_chs at that stage).
        # Order matches self.up_blocks (deepest -> shallow).
        skip_projs = []
        for i in reversed(range(len(down_chs) - 1)):
            skip_chs = down_chs[i]            # channels before DownBlock at level i
            target_chs = down_chs[i + 1]      # decoder in_chs used by UpBlock at that level
            skip_projs.append(nn.Conv2d(skip_chs, target_chs, kernel_size=1))
        self.skip_projs = nn.ModuleList(skip_projs)

        # ---- Final projection back to image channels
        self.out_conv = nn.Conv2d(base_ch, 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:      [B, img_ch, H, W]           (noisy image)
        t:      [B]                          (timestep indices)
        c:      [B] or [B, N_CLASSES]       (class labels or one-hot)
        c_mask: [B, 1]                       (1=use class, 0=ignore; for classifier-free guidance)
        """
        B = x.size(0)

        # ---- Time embedding
        t_emb = self.time_mlp(t)  # [B, deepest_ch, 1, 1]

        # ---- Class embedding (supports indices or one-hot)
        c_emb = self.class_embed(c)                  # [B, c_embed_dim, 1, 1]
        c_emb = self.class_to_ch(c_emb)             # [B, deepest_ch, 1, 1]
        # Apply mask (broadcast)
        if c_mask is not None:
            c_mask_ = c_mask.view(B, 1, 1, 1).to(c_emb.device).type_as(c_emb)
            c_emb = c_emb * c_mask_

        # ---- Initial conv
        h = self.in_conv(x)                          # [B, down_chs[0], H, W]

        # ---- Down path with skip collection
        skips = []
        for i, down in enumerate(self.down_blocks):
            # Save skip BEFORE downsample (so its spatial size is larger; matches UpBlock's expectation)
            skips.append(h)                          # [B, down_chs[i], H_i, W_i]
            h = down(h)                              # [B, down_chs[i+1] (≈), H_i/2, W_i/2]

        # ---- Mid (inject conditioning)
        h = self.mid1(h)
        h = h + t_emb + c_emb                        # broadcast add at bottleneck
        h = self.mid2(h)

        # ---- Up path (reverse order), align skip channels, then fuse
        for idx, up in enumerate(self.up_blocks):
            skip = skips[-(idx + 1)]                 # matching skip (higher-res)
            # project skip channels to match decoder in_chs at this level
            skip_aligned = self.skip_projs[idx](skip)
            h = up(h, skip_aligned)                  # [B, out_chs(level), 2H, 2W]

        # ---- Final projection to image channels (predict noise)
        out = self.out_conv(h)                       # [B, img_ch, H, W]
        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

```
import torch

# Set up device (GPU or CPU) - Added to ensure 'device' is defined
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Set up the noise schedule
n_steps = 100  # How many steps to go from clear image to noise
beta_start = 0.0001  # Starting noise level (small)
beta_end = 0.02      # Ending noise level (larger)

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

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

```
Using device: cuda
```

```

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

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

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

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

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

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

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

    return x_t, noise
```

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

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

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

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

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

```
@torch.no_grad()
def show_noise_progression(image, num_steps=5):
    """
    Visualize how an image gets progressively noisier in the diffusion process.

    Args:
        image (torch.Tensor): [C, H, W] tensor (normalized to [-1, 1] if you used the MNIST transform)
        num_steps (int): number of panels to show (including the original)
    """
    assert image.ndim == 3, "Expected image of shape [C, H, W]"
    img = image.to(device)

    plt.figure(figsize=(3 * num_steps, 3))

    # Helper to rescale for display
    def _display_ready(x):
        x = x.detach().float().cpu()
        if x.min() < 0:  # handle [-1, 1] normalization
            x = (x + 1) / 2
        return x.clamp(0, 1)

    # Panel 1: original
    plt.subplot(1, num_steps, 1)
    if IMG_CH == 1:
        plt.imshow(_display_ready(img)[0], cmap='gray')
    else:
        plt.imshow(_display_ready(img).permute(1, 2, 0))
    plt.title('Original')
    plt.axis('off')

    # Noisier versions
    for i in range(1, num_steps):
        # pick an evenly spaced timestep
        t_idx = int(round((i / (num_steps - 1)) * (n_steps - 1))) if num_steps > 1 else 0
        t_idx = max(0, min(n_steps - 1, t_idx))
        t = torch.tensor([t_idx], device=device)

        noisy, _ = add_noise(img.unsqueeze(0), t)  # [1, C, H, W]
        noisy_img = noisy[0]

        plt.subplot(1, num_steps, i + 1)
        if IMG_CH == 1:
            plt.imshow(_display_ready(noisy_img)[0], cmap='gray')
        else:
            plt.imshow(_display_ready(noisy_img).permute(1, 2, 0))
        pct = int(round(100 * t_idx / (n_steps - 1))) if n_steps > 1 else 0
        plt.title(f't={t_idx}  (~{pct}% 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!

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

# Print model summary
print(f"\n{'='*50}")
print(f"MODEL ARCHITECTURE SUMMARY")
print(f"{'='*50}")
```

```python
print(f"Input resolution: {IMG_SIZE}x{IMG_SIZE}")
print(f"Input channels: {IMG_CH}")
print(f"Time steps: {n_steps}")
print(f"Condition classes: {N_CLASSES}")
print(f"GPU acceleration: {'Yes' if device.type == 'cuda' else 'No'}")

# Validate model parameters and estimate memory requirements
# Hint: Create functions to count parameters and estimate memory usage

# Enter your code here:
# ---- Model stats: params and rough VRAM estimate ----
def count_parameters(m):
    total = sum(p.numel() for p in m.parameters())
    trainable = sum(p.numel() for p in m.parameters() if p.requires_grad)
    return total, trainable

def human_bytes(n):
    for u in ["B","KB","MB","GB","TB"]:
        if n < 1024: return f"{n:.2f} {u}"
        n /= 1024
    return f"{n:.2f} PB"

def estimate_param_memory_bytes(m, dtype_bytes=4):
    # fp32 training rough rule of thumb:
    # weights + grads + Adam m + Adam v ≈ 4 copies
    n_params = sum(p.numel() for p in m.parameters())
    return n_params * 4 * dtype_bytes

total_params, trainable_params = count_parameters(model)
param_mem_bytes = estimate_param_memory_bytes(model)

print(f"\nParameters: {total_params:,} (trainable: {trainable_params:,})")
print(f"~Param+optimizer VRAM (fp32): {human_bytes(param_mem_bytes)}")


# Your code to verify data ranges and integrity
# Hint: Create functions to check data ranges in training and validation data

# Enter your code here:
# ---- Data integrity checks ----
import torch.nn.functional as F
import torch.optim as optim # Added import for optimizers
from torch.utils.data import DataLoader
import numpy as np
import torch # Added import for torch

@torch.no_grad()
def inspect_loader(loader, name="loader", max_batches=3):
    if loader is None:
        print(f"{name}: not available")
        return

    mins, maxs, means, stds = [], [], [], []
    labels = []
    n_samples = 0

    for i, (xb, yb) in enumerate(loader):
        xb = xb.to(device)
        mins.append(float(xb.min()))
        maxs.append(float(xb.max()))
        means.append(float(xb.mean()))
        stds.append(float(xb.std()))
        labels += yb.tolist()
        n_samples += xb.size(0)
        if i + 1 >= max_batches:
            break

    import numpy as np
    if not mins:
        print(f"{name}: (no batches)")
        return

    print(f"\n[{name}] ~{n_samples} samples inspected")
    print(f" value min/max: {min(mins):.3f} / {max(maxs):.3f}")
    print(f" mean ± std:   {np.mean(means):.3f} ± {np.mean(stds):.3f}")
    uniq = sorted(set(labels))
    print(f" labels seen:   {uniq[:20]}{'...' if len(uniq)>20 else ''}")
```

```

# Run checks (if you created these loaders earlier)
inspect_loader(globals().get("train_loader"), "train_loader")
inspect_loader(globals().get("val_loader"),   "val_loader")


# Set up the optimizer with parameters tuned for diffusion models
# Note: Lower learning rates tend to work better for diffusion models
initial_lr = 0.001  # Starting learning rate
weight_decay = 1e-5  # L2 regularization to prevent overfitting

optimizer = optim.Adam( # Changed Adam to optim.Adam
    model.parameters(),
    lr=initial_lr,
    weight_decay=weight_decay
)

# Learning rate scheduler to reduce LR when validation loss plateaus
# This helps fine-tune the model toward the end of training
# Compatible with all PyTorch versions
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer,
    mode="min",          # Reduce LR when monitored value stops decreasing
    factor=0.5,          # Multiply LR by this factor
    patience=5,          # Number of epochs with no improvement
    min_lr=1e-6          # Lower bound on the learning rate
)

# STUDENT EXPERIMENT:
# Try different channel configurations and see how they affect:
# 1. Model size (parameter count)
# 2. Training time
# 3. Generated image quality
#
# Suggestions:
# - Smaller: down_chs=(16, 32, 64)
# - Larger: down_chs=(64, 128, 256, 512)
def step_scheduler(val_loss):
    old_lr = optimizer.param_groups[0]["lr"]
    scheduler.step(val_loss)
    new_lr = optimizer.param_groups[0]["lr"]
    if new_lr != old_lr:
        print(f"⚙️  Learning rate reduced: {old_lr:.6f} → {new_lr:.6f}")
```

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

==================================================
MODEL ARCHITECTURE SUMMARY
==================================================
Input resolution: 28x28
Input channels: 1
Time steps: 100
Condition classes: 10
GPU acceleration: Yes

Parameters: 1,087,901 (trainable: 1,087,901)
~Param+optimizer VRAM (fp32): 16.60 MB

[train_loader] ~192 samples inspected
 value min/max: -1.000 / 1.000
 mean ± std:   -0.734 ± 0.619
 labels seen:   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
val_loader: not available
```

```
# 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:,}")
```

```python
    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
from torchvision.utils import make_grid # Added import for make_grid
import matplotlib.pyplot as plt # Added import for plotting

def generate_samples(model, n_samples=10):
    """
    Generates sample images using the model for visualization during training.
    """
    model.eval()
    with torch.no_grad():
        # Generate digits 0-9 for visualization
        samples = []
        for digit in range(min(n_samples, 10)):
            # Start with random noise
            x = torch.randn(1, IMG_CH, IMG_SIZE, IMG_SIZE).to(device)

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

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

            samples.append(x)

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

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

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

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

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

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

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

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

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

        try:
            emergency_path = path + '.emergency'
            torch.save(model.state_dict(), emergency_path)
            print(f"Emergency save successful: {emergency_path}")
        except:
            print("Emergency save failed. Could not save model.")
```

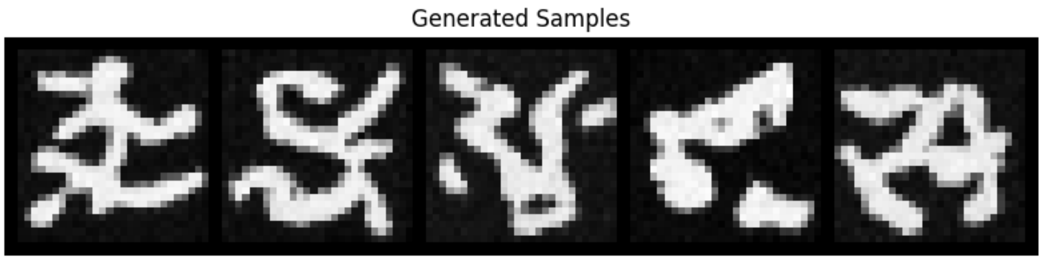
```python
#  Implementation of the training step function
def train_step(x, c):
    """
    Performs a single training step for the diffusion model.
    Returns a scalar tensor loss.
    """
    model.train()
    x = x.to(device, non_blocking=True)
    c = c.to(device, non_blocking=True)

    # 1) Class conditioning (one-hot) + mask
    c_one_hot = F.one_hot(c.long(), N_CLASSES).float().to(device)
    c_mask = torch.ones(c.size(0), 1, device=device)  # [B, 1]

    # 2) Random timesteps for each sample
    t = torch.randint(0, n_steps, (x.size(0),), device=device, dtype=torch.long)

    # 3) Forward diffusion: add noise
    x_t, true_noise = add_noise(x, t)  # true_noise is the target

    # 4) Predict noise
    pred_noise = model(x_t, t, c_one_hot, c_mask)

    # 5) Loss (MSE between predicted and actual noise)
    loss = F.mse_loss(pred_noise, true_noise)

    # 6) Optimize
    optimizer.zero_grad(set_to_none=True)
    loss.backward()
    torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
    optimizer.step()

    return loss.detach()

```

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

# Progress tracking variables
```

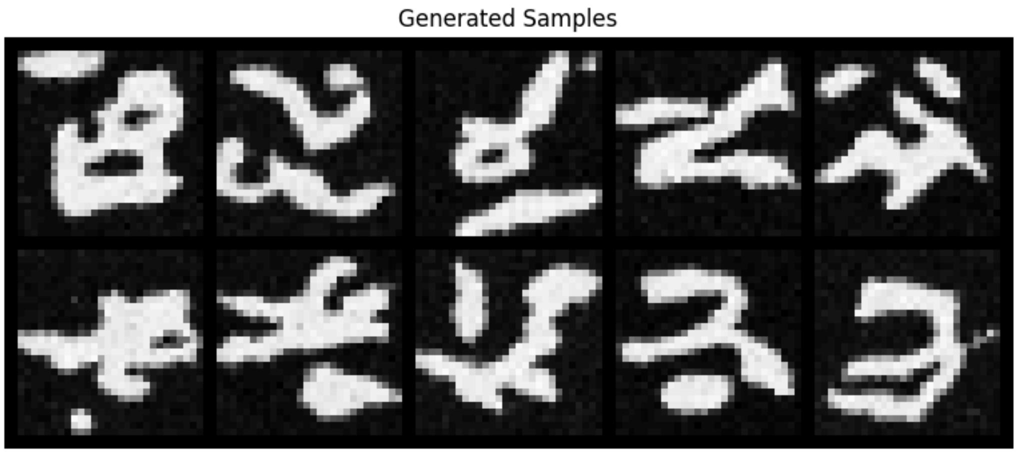
```python
best_loss = float('inf')
train_losses = []
val_losses = []
no_improve_epochs = 0

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

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

# Enter your code here:
import traceback
import matplotlib.pyplot as plt # Added import for plotting

# ---- Validation step (no grad, no optimizer) ----
@torch.no_grad()
def val_step(x, c):
    model.eval()
    x = x.to(device, non_blocking=True)
    c = c.to(device, non_blocking=True)

    c_one_hot = F.one_hot(c.long(), N_CLASSES).float().to(device)
    c_mask = torch.ones(c.size(0), 1, device=device)

    t = torch.randint(0, n_steps, (x.size(0),), device=device, dtype=torch.long)
    x_t, true_noise = add_noise(x, t)
    pred_noise = model(x_t, t, c_one_hot, c_mask)
    return F.mse_loss(pred_noise, true_noise)

# ---- Main training loop ----
early_stopping_patience = 10
gradient_clip_value = 1.0      # (already handled inside train_step)
display_frequency = 100
generate_frequency = 500

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)

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

        for step, (images, labels) in enumerate(train_loader):  # <- use train_loader
            # Single training step (this already zero_grads, backward, clip, and step)
            loss = train_step(images, labels)
            epoch_losses.append(loss.item())

            # Progress
            if (step % display_frequency) == 0:
                print(f"  Step {step}/{len(train_loader)} | Loss: {loss.item():.4f}")

            # Occasional sampling
            if step > 0 and (step % generate_frequency) == 0:
                print("  Generating samples…")
                generate_samples(model, n_samples=5)

        # End of epoch: train summary
        avg_train_loss = sum(epoch_losses) / max(1, 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():
            for val_images, val_labels in test_loader:  # Changed val_loader to test_loader
                vloss = val_step(val_images, val_labels)
                val_epoch_losses.append(vloss.item())

        avg_val_loss = sum(val_epoch_losses) / max(1, len(val_epoch_losses))
        val_losses.append(avg_val_loss)
        print(f"Validation - Epoch {epoch+1} average loss: {avg_val_loss:.4f}")

        # LR scheduling
        # If you defined step_scheduler wrapper earlier, use that; else use scheduler.step directly:
        # step_scheduler(avg_val_loss)
        scheduler.step(avg_val_loss)
        current_lr = optimizer.param_groups[0]['lr']
        print(f"Learning rate: {current_lr:.6f}")

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

        # Save best
        if avg_val_loss < best_loss:
            best_loss = avg_val_loss
            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 losses periodically
        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 KeyboardInterrupt:
    print("\n■ Training interrupted by user (KeyboardInterrupt).")
except Exception as e:
    print("\n❌ Exception during training:")
    print(e)
    traceback.print_exc()
finally:
    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)

    # 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()

```

```
torch.cuda.empty_cache()
```

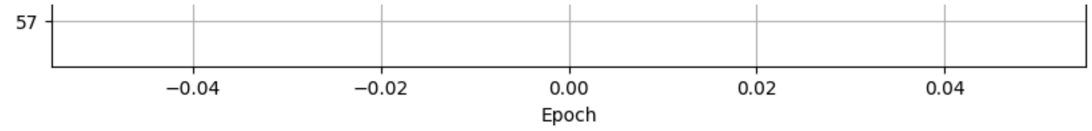
```
==================================================
STARTING TRAINING
==================================================

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

Epoch 1/30
--------------------
  Step 0/938 | Loss: 0.0599
  Step 100/938 | Loss: 0.0669
  Step 200/938 | Loss: 0.0596
  Step 300/938 | Loss: 0.0696
  Step 400/938 | Loss: 0.0685
  Step 500/938 | Loss: 0.0611
  Generating samples…
```



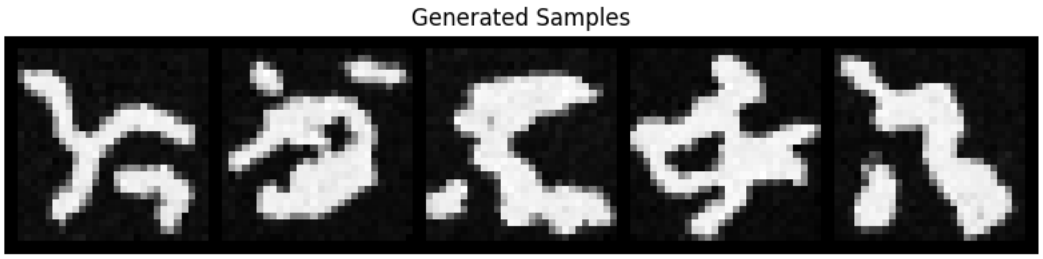
```
  Step 600/938 | Loss: 0.0627
  Step 700/938 | Loss: 0.0614
  Step 800/938 | Loss: 0.0598
  Step 900/938 | Loss: 0.0723

Training - Epoch 1 average loss: 0.0642
Running validation…
Validation - Epoch 1 average loss: 0.0642
Learning rate: 0.001000

Generating samples for visual progress check…
```



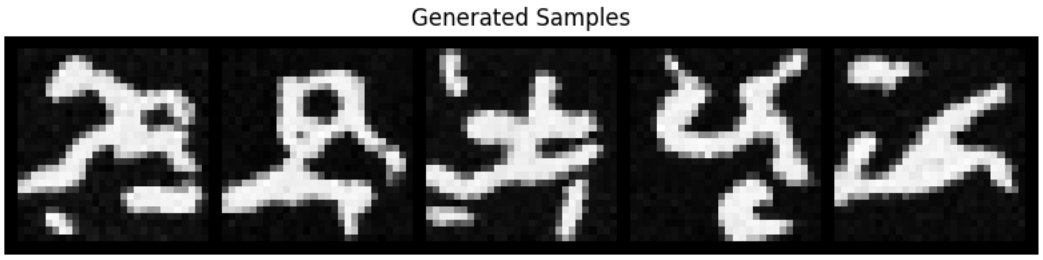
```
Model successfully saved to best_diffusion_model.pt
✓ New best model saved! (Val Loss: 0.0642)
```



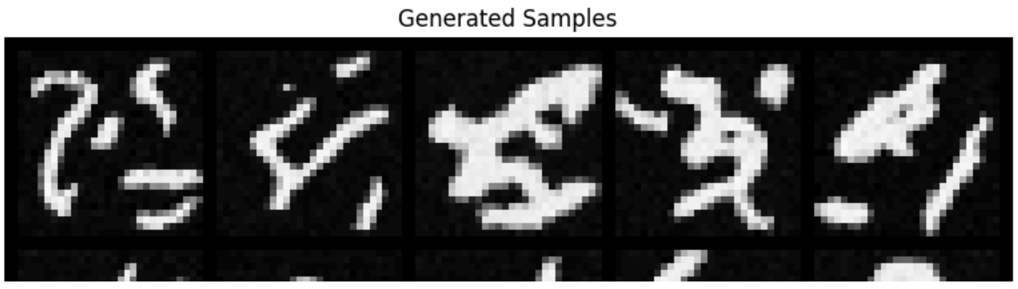
```
Epoch 2/30
--------------------
  Step 0/938 | Loss: 0.0611
  Step 100/938 | Loss: 0.0593
  Step 200/938 | Loss: 0.0684
  Step 300/938 | Loss: 0.0714
  Step 400/938 | Loss: 0.0672
  Step 500/938 | Loss: 0.0659
  Generating samples…
```



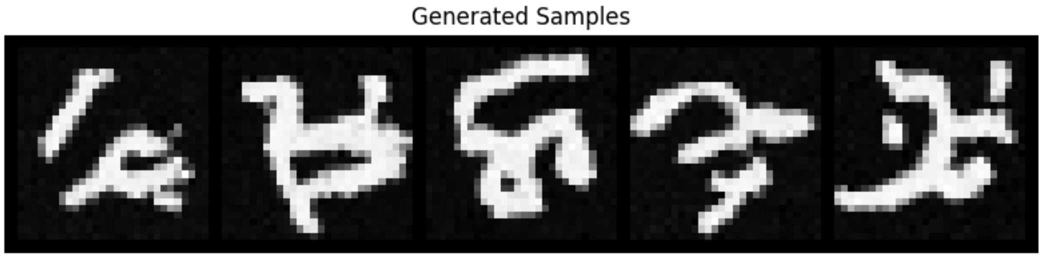
```
  Step 600/938 | Loss: 0.0650
  Step 700/938 | Loss: 0.0658
  Step 800/938 | Loss: 0.0589
  Step 900/938 | Loss: 0.0544

Training - Epoch 2 average loss: 0.0627
Running validation…
Validation - Epoch 2 average loss: 0.0619
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: 0.0619)

Epoch 3/30
--------------------
  Step 0/938 | Loss: 0.0565
  Step 100/938 | Loss: 0.0674
  Step 200/938 | Loss: 0.0612
  Step 300/938 | Loss: 0.0697
  Step 400/938 | Loss: 0.0553
  Step 500/938 | Loss: 0.0533
  Generating samples…
```



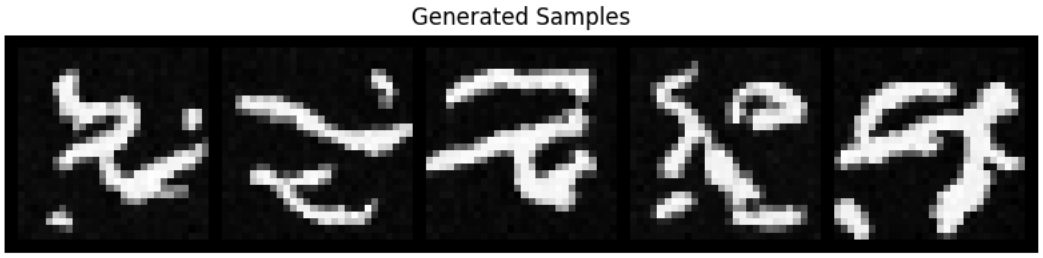
```
  Step 600/938 | Loss: 0.0606
  Step 700/938 | Loss: 0.0544
  Step 800/938 | Loss: 0.0562
  Step 900/938 | Loss: 0.0657

Training - Epoch 3 average loss: 0.0618
Running validation…
Validation - Epoch 3 average loss: 0.0621
Learning rate: 0.001000

Generating samples for visual progress check…
```



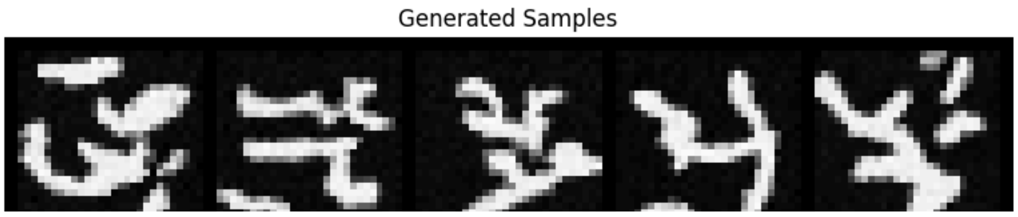
```
No improvement for 1/10 epochs

Epoch 4/30
--------------------
  Step 0/938 | Loss: 0.0530
  Step 100/938 | Loss: 0.0598
  Step 200/938 | Loss: 0.0754
  Step 300/938 | Loss: 0.0549
  Step 400/938 | Loss: 0.0534
  Step 500/938 | Loss: 0.0560
  Generating samples…
```

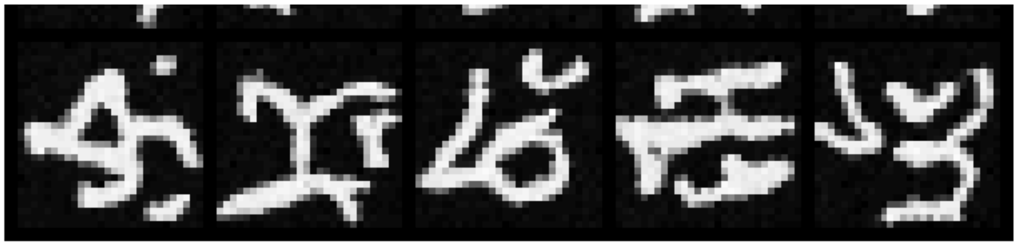


```
  Step 600/938 | Loss: 0.0717
  Step 700/938 | Loss: 0.0568
  Step 800/938 | Loss: 0.0667
  Step 900/938 | Loss: 0.0571

Training - Epoch 4 average loss: 0.0606
Running validation…
Validation - Epoch 4 average loss: 0.0614
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: 0.0614)

Epoch 5/30
--------------------
  Step 0/938 | Loss: 0.0646
  Step 100/938 | Loss: 0.0594
  Step 200/938 | Loss: 0.0591
  Step 300/938 | Loss: 0.0745
  Step 400/938 | Loss: 0.0526
  Step 500/938 | Loss: 0.0646
  Generating samples…
```



```
  Step 600/938 | Loss: 0.0588
  Step 700/938 | Loss: 0.0560
  Step 800/938 | Loss: 0.0671
  Step 900/938 | Loss: 0.0521

Training - Epoch 5 average loss: 0.0601
Running validation…
Validation - Epoch 5 average loss: 0.0607
Learning rate: 0.001000

Generating samples for visual progress check…
```

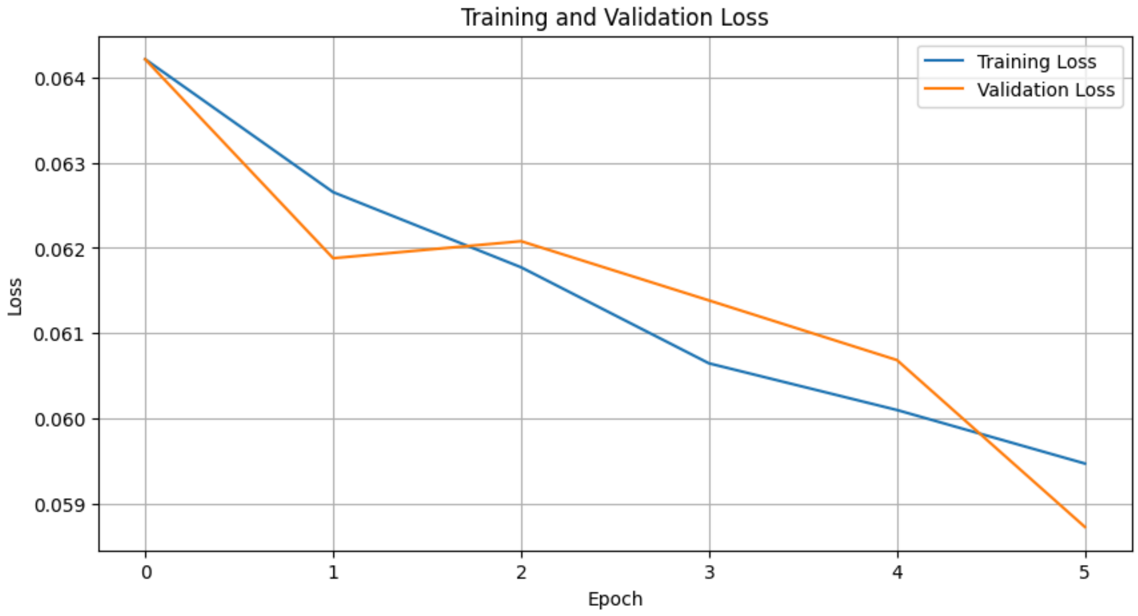




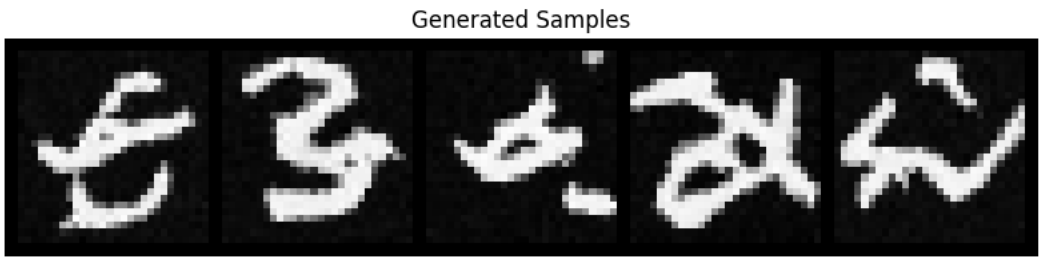
```
Created backup at best_diffusion_model.pt.backup
Model successfully saved to best_diffusion_model.pt
✓ New best model saved! (Val Loss: 0.0607)

Epoch 6/30
--------------------
  Step 0/938 | Loss: 0.0705
  Step 100/938 | Loss: 0.0687
  Step 200/938 | Loss: 0.0645
  Step 300/938 | Loss: 0.0519
  Step 400/938 | Loss: 0.0547
  Step 500/938 | Loss: 0.0601
  Generating samples…
```

Generated Samples

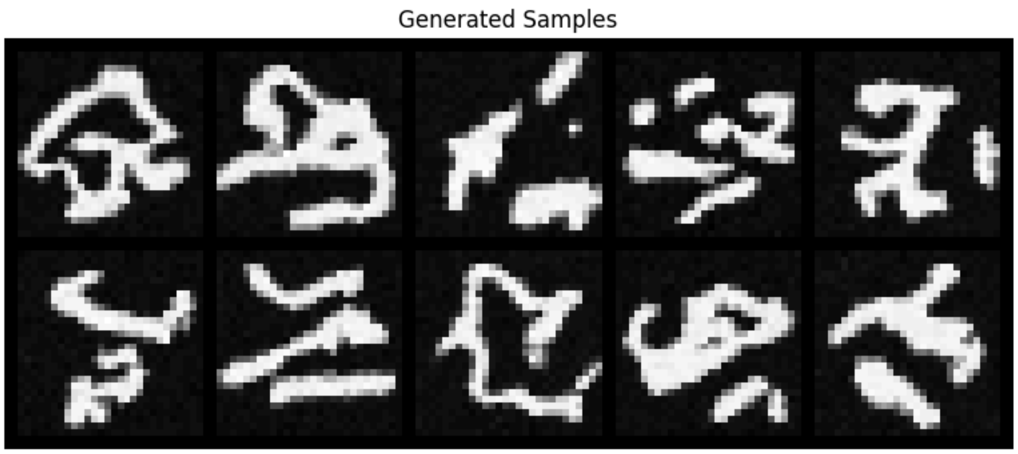
```
  Step 600/938 | Loss: 0.0672
  Step 700/938 | Loss: 0.0584
  Step 800/938 | Loss: 0.0628
  Step 900/938 | Loss: 0.0558

Training - Epoch 6 average loss: 0.0595
Running validation…
Validation - Epoch 6 average loss: 0.0587
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: 0.0587)
```



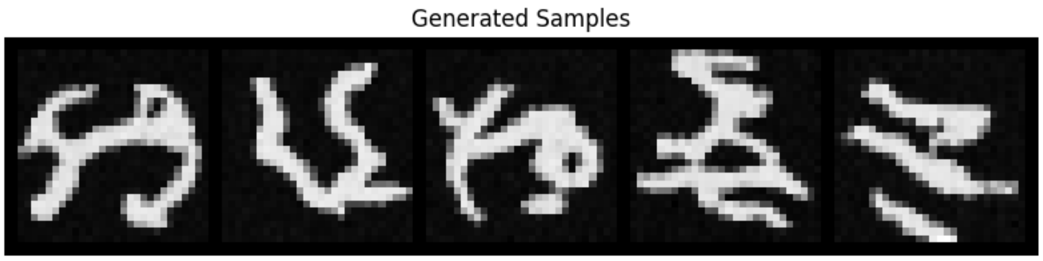
```
Epoch 7/30
--------------------
  Step 0/938 | Loss: 0.0538
  Step 100/938 | Loss: 0.0489
  Step 200/938 | Loss: 0.0549
  Step 300/938 | Loss: 0.0553
  Step 400/938 | Loss: 0.0623
  Step 500/938 | Loss: 0.0623
  Generating samples…
```



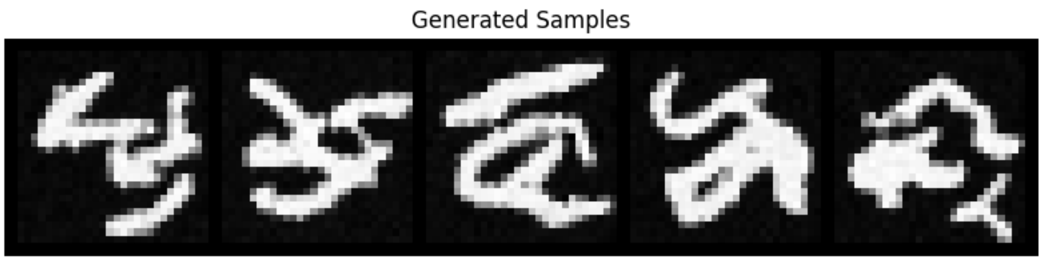
```
  Step 600/938 | Loss: 0.0627
  Step 700/938 | Loss: 0.0550
  Step 800/938 | Loss: 0.0612
  Step 900/938 | Loss: 0.0580

Training - Epoch 7 average loss: 0.0591
Running validation…
Validation - Epoch 7 average loss: 0.0587
Learning rate: 0.001000

Generating samples for visual progress check…
```



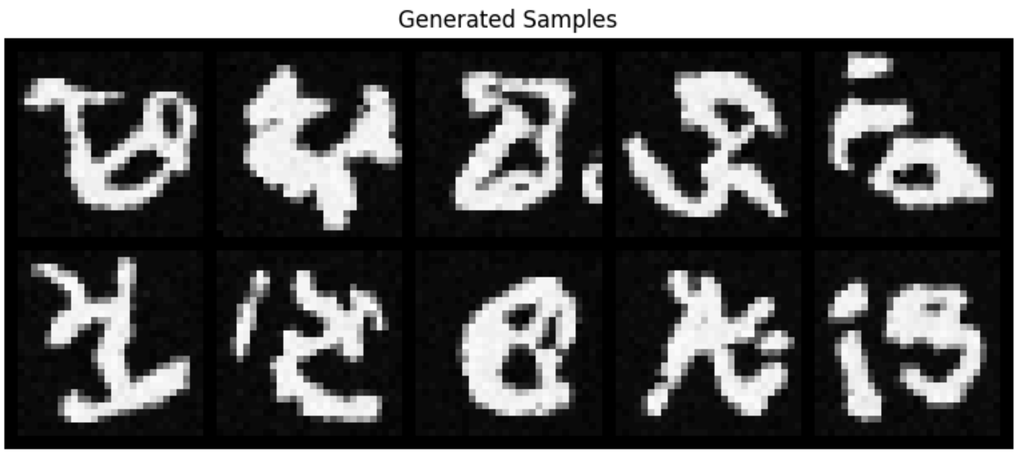
```
Created backup at best_diffusion_model.pt.backup
Model successfully saved to best_diffusion_model.pt
✓ New best model saved! (Val Loss: 0.0587)

Epoch 8/30
--------------------
  Step 0/938 | Loss: 0.0512
  Step 100/938 | Loss: 0.0553
  Step 200/938 | Loss: 0.0610
  Step 300/938 | Loss: 0.0547
  Step 400/938 | Loss: 0.0525
  Step 500/938 | Loss: 0.0592
  Generating samples…
```



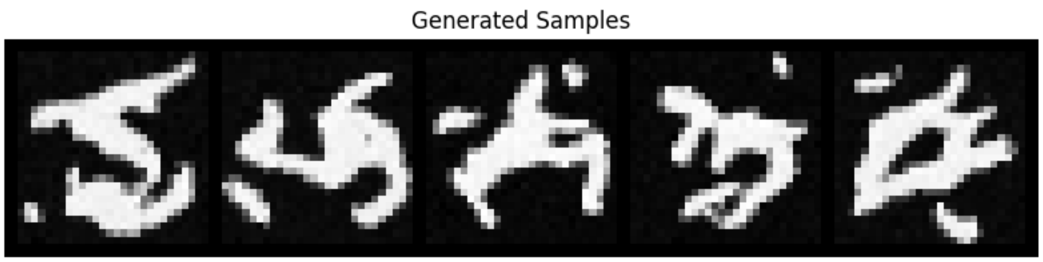
```
  Step 600/938 | Loss: 0.0573
  Step 700/938 | Loss: 0.0741
  Step 800/938 | Loss: 0.0577
  Step 900/938 | Loss: 0.0496

Training - Epoch 8 average loss: 0.0589
Running validation…
Validation - Epoch 8 average loss: 0.0582
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: 0.0582)

Epoch 9/30
--------------------
  Step 0/938 | Loss: 0.0552
  Step 100/938 | Loss: 0.0542
  Step 200/938 | Loss: 0.0585
  Step 300/938 | Loss: 0.0530
  Step 400/938 | Loss: 0.0498
  Step 500/938 | Loss: 0.0537
  Generating samples…
```



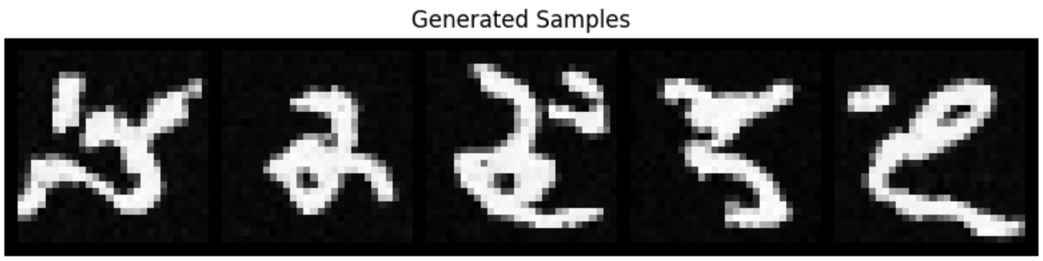
```
  Step 600/938 | Loss: 0.0589
  Step 700/938 | Loss: 0.0508
  Step 800/938 | Loss: 0.0605
  Step 900/938 | Loss: 0.0593

Training - Epoch 9 average loss: 0.0589
Running validation…
Validation - Epoch 9 average loss: 0.0597
Learning rate: 0.001000

Generating samples for visual progress check…
```



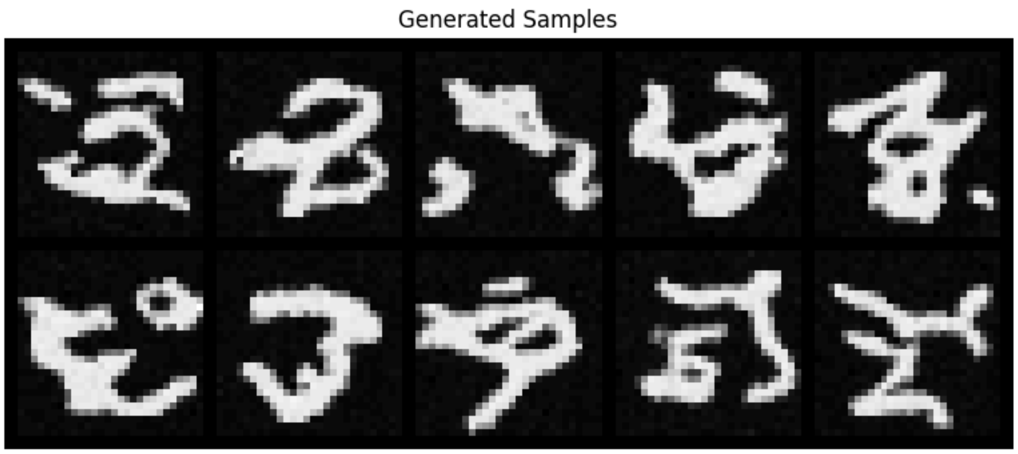
```
No improvement for 1/10 epochs

Epoch 10/30
--------------------
  Step 0/938 | Loss: 0.0552
  Step 100/938 | Loss: 0.0540
  Step 200/938 | Loss: 0.0626
  Step 300/938 | Loss: 0.0516
  Step 400/938 | Loss: 0.0673
  Step 500/938 | Loss: 0.0548
  Generating samples…
```



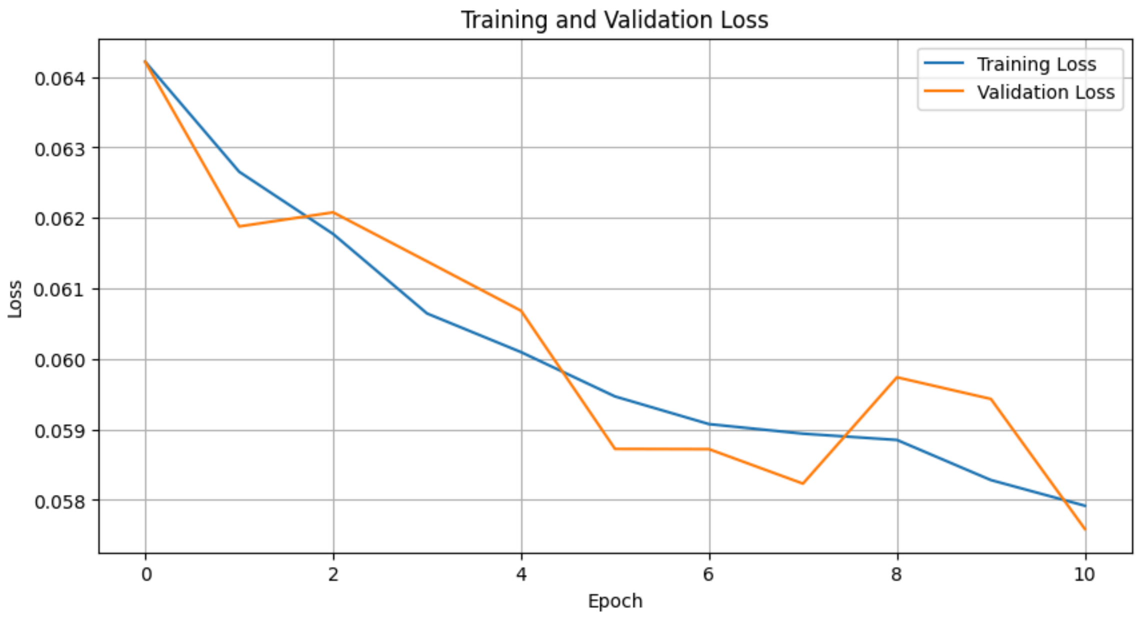
```
  Step 600/938 | Loss: 0.0577
  Step 700/938 | Loss: 0.0611
  Step 800/938 | Loss: 0.0616
  Step 900/938 | Loss: 0.0613

Training - Epoch 10 average loss: 0.0583
Running validation…
Validation - Epoch 10 average loss: 0.0594
Learning rate: 0.001000
No improvement for 2/10 epochs

Epoch 11/30
--------------------
  Step 0/938 | Loss: 0.0512
  Step 100/938 | Loss: 0.0526
  Step 200/938 | Loss: 0.0556
  Step 300/938 | Loss: 0.0545
  Step 400/938 | Loss: 0.0536
  Step 500/938 | Loss: 0.0590
  Generating samples…
```



```
  Step 600/938 | Loss: 0.0691
  Step 700/938 | Loss: 0.0659
  Step 800/938 | Loss: 0.0568
  Step 900/938 | Loss: 0.0495

Training - Epoch 11 average loss: 0.0579
Running validation…
Validation - Epoch 11 average loss: 0.0576
Learning rate: 0.001000

Generating samples for visual progress check…
```

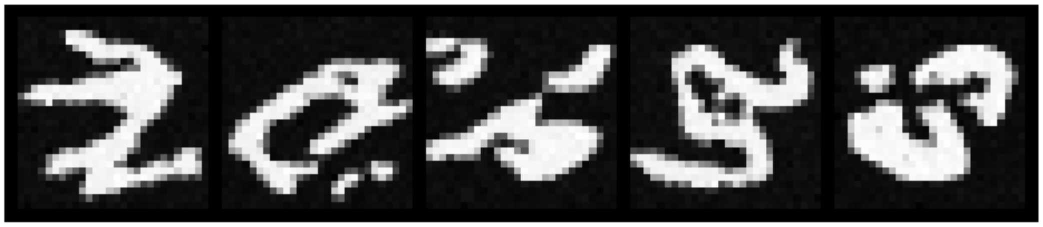


```
Created backup at best_diffusion_model.pt.backup
Model successfully saved to best_diffusion_model.pt
✓ New best model saved! (Val Loss: 0.0576)
```



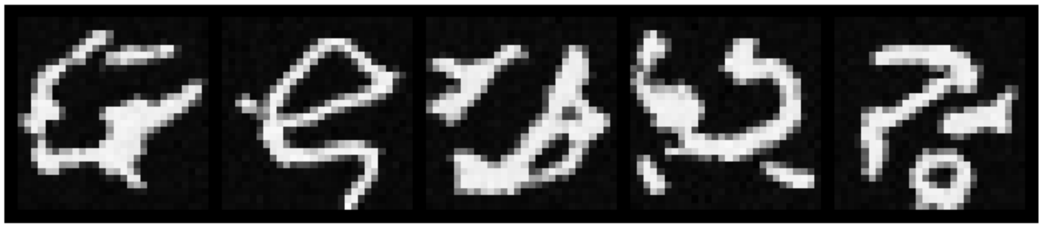
```
Epoch 12/30
--------------------
  Step 0/938 | Loss: 0.0636
  Step 100/938 | Loss: 0.0568
  Step 200/938 | Loss: 0.0634
  Step 300/938 | Loss: 0.0553
  Step 400/938 | Loss: 0.0579
  Step 500/938 | Loss: 0.0564
  Generating samples…
```

Generated Samples



```
  Step 600/938 | Loss: 0.0623
  Step 700/938 | Loss: 0.0573
  Step 800/938 | Loss: 0.0531
  Step 900/938 | Loss: 0.0625

Training - Epoch 12 average loss: 0.0579
Running validation…
Validation - Epoch 12 average loss: 0.0575
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: 0.0575)

Epoch 13/30
--------------------
  Step 0/938 | Loss: 0.0495
  Step 100/938 | Loss: 0.0525
  Step 200/938 | Loss: 0.0618
  Step 300/938 | Loss: 0.0503
  Step 400/938 | Loss: 0.0591
  Step 500/938 | Loss: 0.0516
  Generating samples…
```

Generated Samples



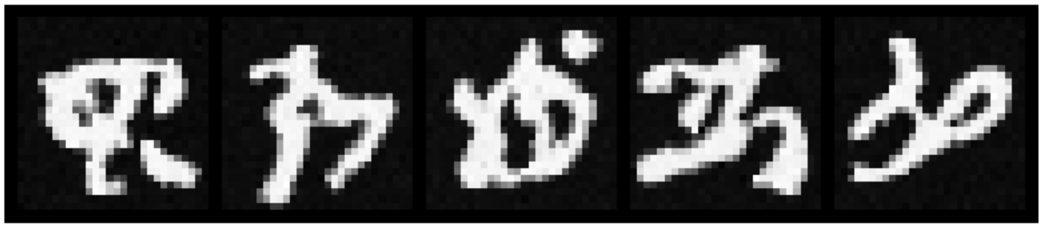
```
  Step 600/938 | Loss: 0.0553
  Step 700/938 | Loss: 0.0653
  Step 800/938 | Loss: 0.0605
  Step 900/938 | Loss: 0.0582

Training - Epoch 13 average loss: 0.0576
Running validation…
Validation - Epoch 13 average loss: 0.0574
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: 0.0574)
```

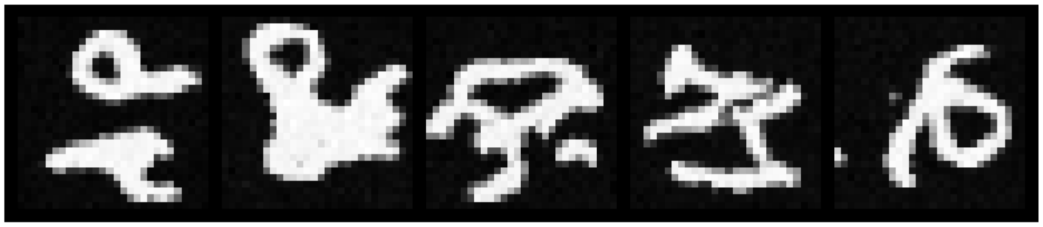
```
Epoch 14/30
--------------------
  Step 0/938 | Loss: 0.0593
  Step 100/938 | Loss: 0.0670
  Step 200/938 | Loss: 0.0548
  Step 300/938 | Loss: 0.0473
  Step 400/938 | Loss: 0.0561
  Step 500/938 | Loss: 0.0650
  Generating samples…
```

Generated Samples



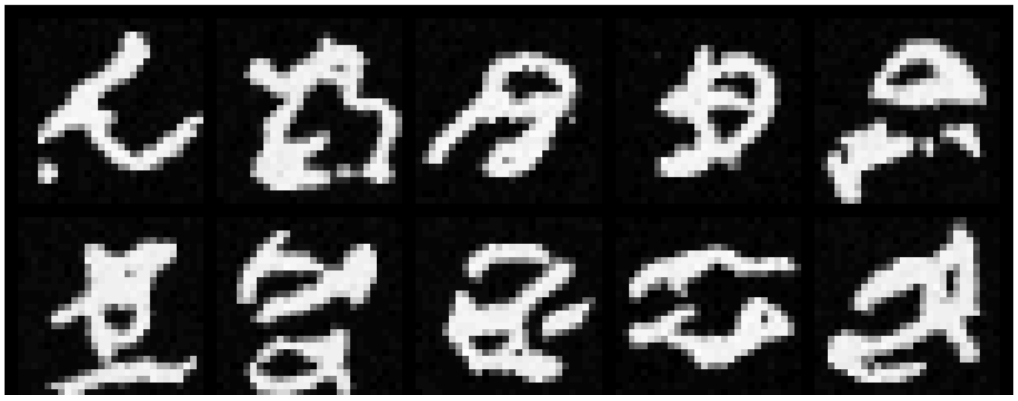
```
  Step 600/938 | Loss: 0.0577
  Step 700/938 | Loss: 0.0570
  Step 800/938 | Loss: 0.0594
  Step 900/938 | Loss: 0.0663

Training - Epoch 14 average loss: 0.0573
Running validation…
Validation - Epoch 14 average loss: 0.0570
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: 0.0570)

Epoch 15/30
--------------------
  Step 0/938 | Loss: 0.0534
  Step 100/938 | Loss: 0.0665
  Step 200/938 | Loss: 0.0606
  Step 300/938 | Loss: 0.0614
  Step 400/938 | Loss: 0.0475
  Step 500/938 | Loss: 0.0569
  Generating samples…
```

Generated Samples

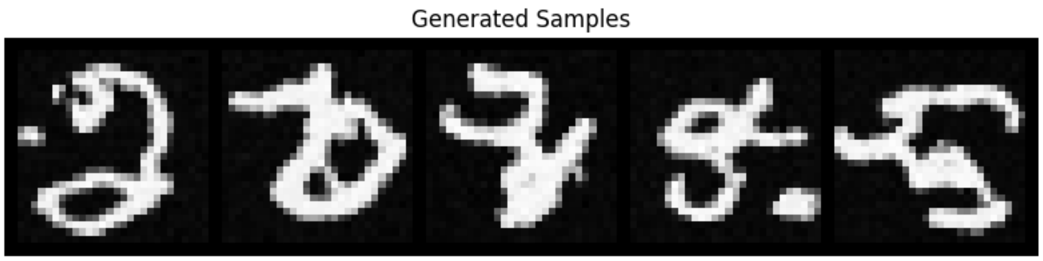


```
  Step 600/938 | Loss: 0.0561
  Step 700/938 | Loss: 0.0630
  Step 800/938 | Loss: 0.0503
  Step 900/938 | Loss: 0.0543

Training - Epoch 15 average loss: 0.0574
Running validation…
Validation - Epoch 15 average loss: 0.0567
Learning rate: 0.001000

Generating samples for visual progress check…
```

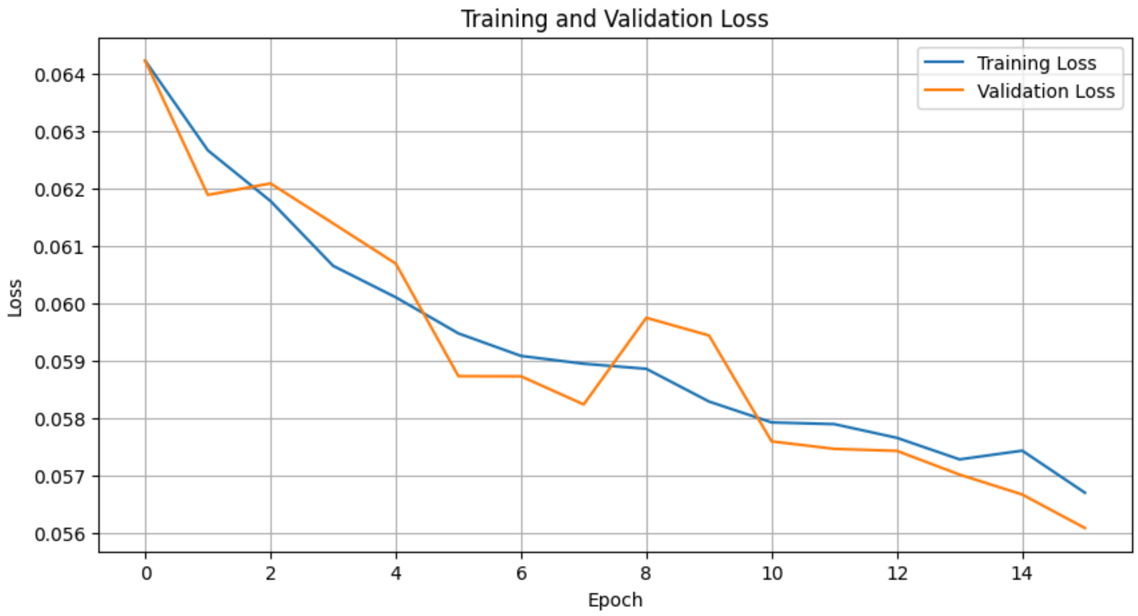
Generated Samples



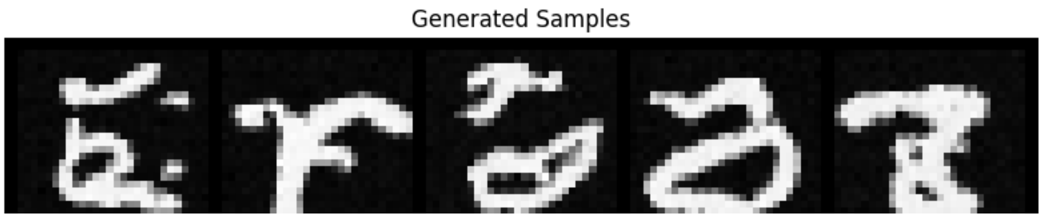
```
Created backup at best_diffusion_model.pt.backup
Model successfully saved to best_diffusion_model.pt
✓ New best model saved! (Val Loss: 0.0567)

Epoch 16/30
--------------------
  Step 0/938 | Loss: 0.0647
  Step 100/938 | Loss: 0.0561
  Step 200/938 | Loss: 0.0565
  Step 300/938 | Loss: 0.0669
  Step 400/938 | Loss: 0.0562
  Step 500/938 | Loss: 0.0500
  Generating samples…
```



```
  Step 600/938 | Loss: 0.0591
  Step 700/938 | Loss: 0.0469
  Step 800/938 | Loss: 0.0580
  Step 900/938 | Loss: 0.0587

Training - Epoch 16 average loss: 0.0567
Running validation…
Validation - Epoch 16 average loss: 0.0561
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: 0.0561)
```

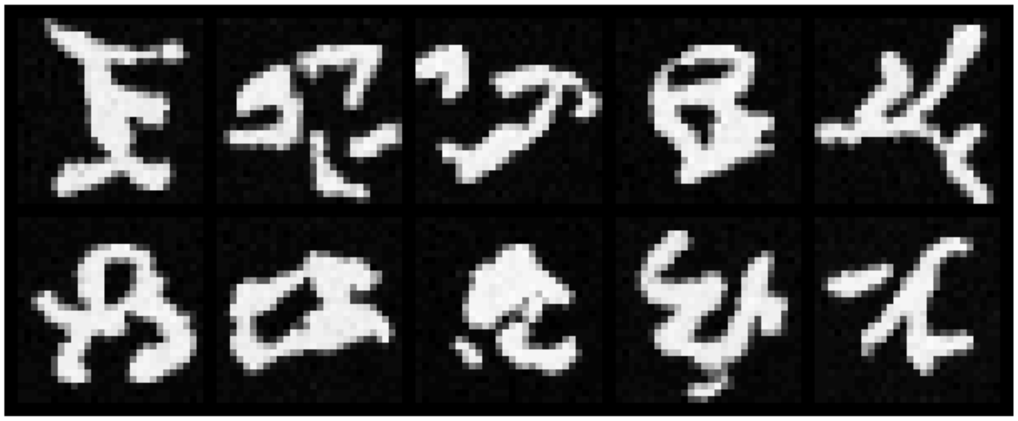


```
Epoch 17/30
--------------------
  Step 0/938 | Loss: 0.0572
  Step 100/938 | Loss: 0.0552
  Step 200/938 | Loss: 0.0614
  Step 300/938 | Loss: 0.0618
  Step 400/938 | Loss: 0.0480
  Step 500/938 | Loss: 0.0574
  Generating samples…
```



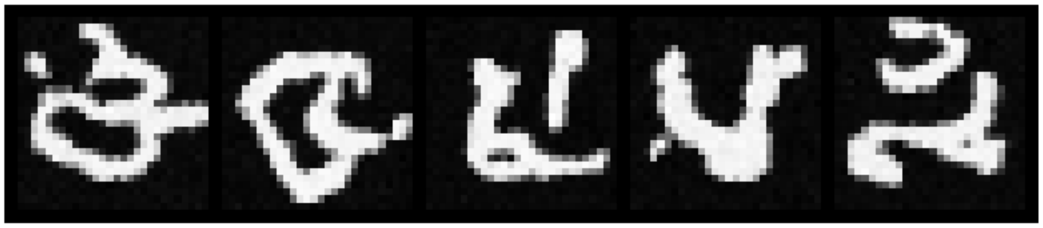
```
  Step 600/938 | Loss: 0.0556
  Step 700/938 | Loss: 0.0572
  Step 800/938 | Loss: 0.0512
  Step 900/938 | Loss: 0.0542

Training - Epoch 17 average loss: 0.0569
Running validation…
Validation - Epoch 17 average loss: 0.0561
Learning rate: 0.001000

Generating samples for visual progress check…
```

Generated Samples



```
No improvement for 1/10 epochs

Epoch 18/30
--------------------
  Step 0/938 | Loss: 0.0635
  Step 100/938 | Loss: 0.0544
  Step 200/938 | Loss: 0.0574
  Step 300/938 | Loss: 0.0565
  Step 400/938 | Loss: 0.0536
  Step 500/938 | Loss: 0.0528
  Generating samples…
```

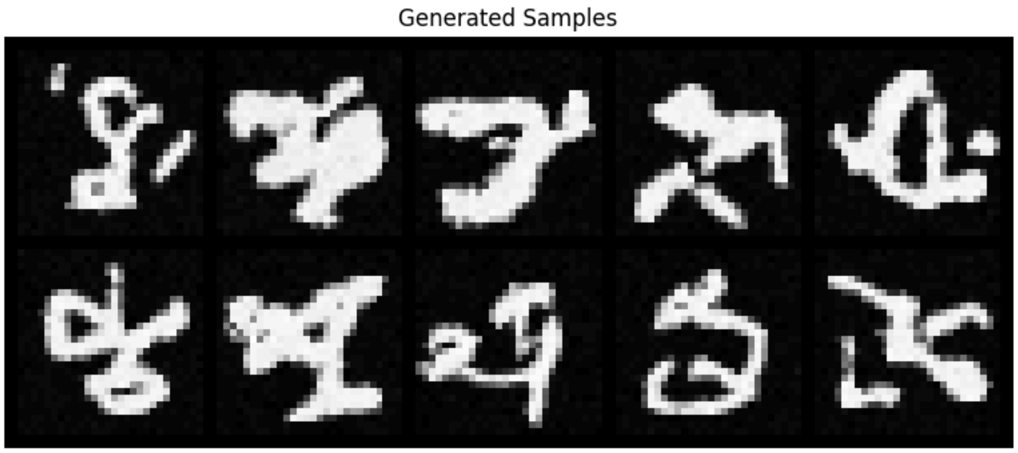
Generated Samples



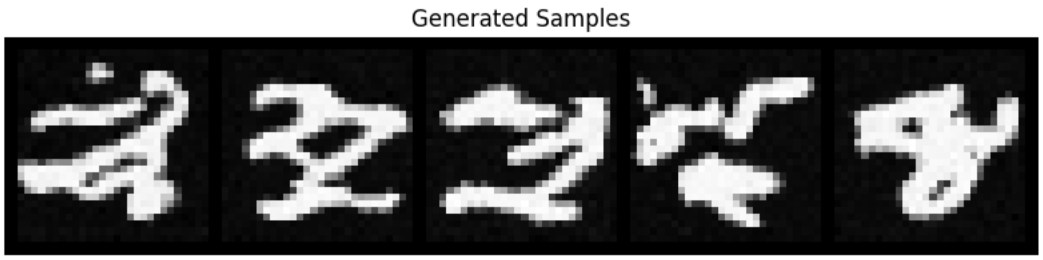
```
  Step 600/938 | Loss: 0.0523
  Step 700/938 | Loss: 0.0540
  Step 800/938 | Loss: 0.0591
  Step 900/938 | Loss: 0.0598

Training - Epoch 18 average loss: 0.0569
Running validation…
Validation - Epoch 18 average loss: 0.0556
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: 0.0556)

Epoch 19/30
--------------------
  Step 0/938 | Loss: 0.0636
  Step 100/938 | Loss: 0.0574
  Step 200/938 | Loss: 0.0602
  Step 300/938 | Loss: 0.0615
  Step 400/938 | Loss: 0.0606
  Step 500/938 | Loss: 0.0538
  Generating samples…
```

Generated Samples

```
  Step 600/938 | Loss: 0.0550
  Step 700/938 | Loss: 0.0523
  Step 800/938 | Loss: 0.0533
  Step 900/938 | Loss: 0.0572

Training - Epoch 19 average loss: 0.0568
Running validation…
Validation - Epoch 19 average loss: 0.0561
Learning rate: 0.001000

Generating samples for visual progress check…
```



```
No improvement for 1/10 epochs

Epoch 20/30
--------------------
  Step 0/938 | Loss: 0.0506
  Step 100/938 | Loss: 0.0621
  Step 200/938 | Loss: 0.0486
  Step 300/938 | Loss: 0.0485
  Step 400/938 | Loss: 0.0557
  Step 500/938 | Loss: 0.0467
  Generating samples…
```



```
  Step 600/938 | Loss: 0.0616
  Step 700/938 | Loss: 0.0552
  Step 800/938 | Loss: 0.0648
  Step 900/938 | Loss: 0.0544

Training - Epoch 20 average loss: 0.0565
Running validation…
Validation - Epoch 20 average loss: 0.0554
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: 0.0554)

Epoch 21/30
--------------------
  Step 0/938 | Loss: 0.0558
  Step 100/938 | Loss: 0.0606
  Step 200/938 | Loss: 0.0575
  Step 300/938 | Loss: 0.0463
  Step 400/938 | Loss: 0.0482
  Step 500/938 | Loss: 0.0618
  Generating samples…
```