

## 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.

```
1 # Step 1: Install required packages
2 %pip install einops
3 print("Package installation complete.")
4
5 # Step 2: Import libraries
6 # --- Core PyTorch libraries ---
7 import torch # Main deep learning framework
8 import torch.nn.functional as F # Neural network functions like activation functions
9 import torch.nn as nn # Neural network building blocks (layers)
10 from torch.optim import Adam # Optimization algorithm for training
11
12 # --- Data handling ---
13 from torch.utils.data import Dataset, DataLoader # For organizing and loading our data
14 import torchvision # Library for computer vision datasets and models
15 import torchvision.transforms as transforms # For preprocessing images
16
17 # --- Tensor manipulation ---
18 import random # For random operations
```

```

19 from einops.layers.torch import Rearrange # For reshaping tensors in neural networks
20 from einops import rearrange # For elegant tensor reshaping operations
21 import numpy as np # For numerical operations on arrays
22
23 # --- System utilities ---
24 import os # For operating system interactions (used for CPU count)
25
26 # --- Visualization tools ---
27 import matplotlib.pyplot as plt # For plotting images and graphs
28 from PIL import Image # For image processing
29 from torchvision.utils import save_image, make_grid # For saving and displaying image grids
30
31 # Step 3: Set up device (GPU or CPU)
32 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
33 print(f"We'll be using: {device}")
34
35 # Check if we're actually using GPU (for students to verify)
36 if device.type == "cuda":
37     print(f"GPU name: {torch.cuda.get_device_name(0)}")
38     print(f"GPU memory: {torch.cuda.get_device_properties(0).total_memory / 1e9:.2f} GB")
39 else:
40     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

```

1 # Step 4: Set random seeds for reproducibility
2 # Diffusion models are sensitive to initialization, so reproducible results help with debugging
3 SEED = 42 # Universal seed value for reproducibility
4 torch.manual_seed(SEED) # PyTorch random number generator
5 np.random.seed(SEED) # NumPy random number generator
6 random.seed(SEED) # Python's built-in random number generator
7
8 print(f"Random seeds set to {SEED} for reproducible results")
9
10 # Configure CUDA for GPU operations if available
11 if torch.cuda.is_available():
12     torch.cuda.manual_seed(SEED) # GPU random number generator
13     torch.cuda.manual_seed_all(SEED) # All GPUs random number generator
14
15     # Ensure deterministic GPU operations
16     # Note: This slightly reduces performance but ensures results are reproducible
17     torch.backends.cudnn.deterministic = True
18     torch.backends.cudnn.benchmark = False
19
20     try:
21         # Check available GPU memory
22         gpu_memory = torch.cuda.get_device_properties(0).total_memory / 1e9 # Convert to GB
23         print(f"Available GPU Memory: {gpu_memory:.1f} GB")
24
25         # Add recommendation based on memory
26         if gpu_memory < 4:
27             print("Warning: Low GPU memory. Consider reducing batch size if you encounter OOM errors.")
28     except Exception as e:
29         print(f"Could not check GPU memory: {e}")
30 else:
31     print("No GPU detected. Training will be much slower on CPU.")
32     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 es 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.

```

1 =====
2 # SECTION 2: DATASET SELECTION AND CONFIGURATION
3 =====
4 # STUDENT INSTRUCTIONS:
5 # 1. Choose ONE dataset option based on your available GPU memory
6 # 2. Uncomment ONLY ONE dataset section below
7 # 3. Make sure all other dataset sections remain commented out
8
9 -----
10 # OPTION 1: MNIST (Basic - 2GB GPU)
11 -----
12 # Recommended for: Free Colab or basic GPU
13 # Memory needed: ~2GB GPU
14 # Training time: ~15-30 minutes
15
16 IMG_SIZE = 28
17 IMG_CH = 1
18 N_CLASSES = 10
19 BATCH_SIZE = 64
20 EPOCHS = 30
21
22 transform = transforms.Compose([
23     transforms.ToTensor(),
24     transforms.Normalize((0.5,), (0.5,))
25 ])
26
27 # Your code to load the MNIST dataset
28 # Hint: Use torchvision.datasets.MNIST with root='./data', train=True,
29 #         transform=transform, and download=True
30 # Then print a success message
31
32 # Enter your code here:
33
34
35 -----
36 # OPTION 2: FASHION-MNIST (Intermediate - 2GB GPU)

```

```

50 # OPTION 2: Fashion-MNIST (intermediate - 2GB GPU)
51 #-----
52 # Uncomment this section to use Fashion-MNIST instead
53 """
54     40 IMG_SIZE = 28
55     41 IMG_CH = 1
56     42 N_CLASSES = 10
57     43 BATCH_SIZE = 64
58     44 EPOCHS = 30
59 """
60 #-----
61 # Your code to load the Fashion-MNIST dataset
62 # Hint: Very similar to MNIST but use torchvision.datasets.FashionMNIST
63 """
64 # Enter your code here:
65 """
66 """
67 """
68 """
69 # Your code to create the transform and load CIFAR-10
70 # Hint: Use transforms.Normalize with RGB means and stds ((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
71 # Then load torchvision.datasets.CIFAR10
72 """
73 # Enter your code here:
74 """
75 """

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

```

```

1 #=====
2 # SECTION 2: DATASET SELECTION AND CONFIGURATION
3 #=====
4 # STUDENT INSTRUCTIONS:
5 # 1. Choose ONE dataset option based on your available GPU memory
6 # 2. Uncomment ONLY ONE dataset section below
7 # 3. Make sure all other dataset sections remain commented out
8 """
9 #
10 # OPTION 1: MNIST (Basic - 2GB GPU)
11 #
12 # Recommended for: Free Colab or basic GPU
13 # Memory needed: ~2GB GPU
14 # Training time: ~15-30 minutes
15
16 IMG_SIZE = 28
17 IMG_CH = 1
18 N_CLASSES = 10
19 BATCH_SIZE = 64
20 EPOCHS = 30
21
22 transform = transforms.Compose([
23     transforms.ToTensor(),
24     transforms.Normalize((0.5,), (0.5,)))
25 ])
26
27 # Your code to load the MNIST dataset
28 # Hint: Use torchvision.datasets.MNIST with root='./data', train=True,
29 #         transform=transform, and download=True
30 # Then print a success message
31
32 # Enter your code here:

```

```

33 # (This assumes you have already run:
34 # import torchvision
35 # import torchvision.transforms as transforms
36 # )
37
38 train_dataset = torchvision.datasets.MNIST(
39     root='./data',
40     train=True,
41     transform=transform,
42     download=True
43 )
44
45 print("✅ Successfully loaded MNIST training dataset.")
46 print(f" - Dataset size: {len(train_dataset)} samples")
47 print(f" - Image config: {IMG_SIZE}x{IMG_SIZE}x{IMG_CH}")
48
49
50 #-----
51 # OPTION 2: Fashion-MNIST (Intermediate - 2GB GPU)
52 #-----
53 # Uncomment this section to use Fashion-MNIST instead
54 """
55 IMG_SIZE = 28
56 IMG_CH = 1
57 N_CLASSES = 10
58 BATCH_SIZE = 64
59 EPOCHS = 30
60
61 transform = transforms.Compose([
62     transforms.ToTensor(),
63     transforms.Normalize((0.5,), (0.5,))
64 ])
65
66 # Your code to load the Fashion-MNIST dataset
67 # Hint: Very similar to MNIST but use torchvision.datasets.FashionMNIST
68
69 # Enter your code here:
70
71 """
72
73 #-----
74 # OPTION 3: CIFAR-10 (Advanced - 4GB+ GPU)
75 #-----
76 # Uncomment this section to use CIFAR-10 instead
77 """
78 IMG_SIZE = 32
79 IMG_CH = 3
80 N_CLASSES = 10
81 BATCH_SIZE = 32 # Reduced batch size for memory
82 EPOCHS = 50 # More epochs for complex data
83
84 # Your code to create the transform and load CIFAR-10
85 # Hint: Use transforms.Normalize with RGB means and stds ((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
86 # Then load torchvision.datasets.CIFAR10
87
88 # Enter your code here:
89
90 """

```

100% |██████████| 9.91M/9.91M [00:02<00:00, 4.79MB/s]  
100% |██████████| 28.9k/28.9k [00:00<00:00, 132kB/s]  
100% |██████████| 1.65M/1.65M [00:01<00:00, 1.24MB/s]  
100% |██████████| 4.54k/4.54k [00:00<00:00, 13.5MB/s] ✅ Successfully loaded MNIST training dataset.  
- Dataset size: 60000 samples  
- Image config: 28x28x1

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

```

1 #Validating Dataset Selection
2 #Let's add code to validate that a dataset was selected
3 # and check if your GPU has enough memory:
4
5 # Validate dataset selection
6 # *** I've changed 'dataset' to 'train_dataset' to match your variable ***
7 if 'train_dataset' not in locals():

```

```

8     raise ValueError("'''")
9     ✘ ERROR: No dataset selected! Please uncomment exactly one dataset option.
10    (Note: I'm checking for 'train_dataset', which you loaded successfully.)
11    Available options:
12    1. MNIST (Basic) - 2GB GPU
13    2. Fashion-MNIST (Intermediate) - 2GB GPU
14    3. CIFAR-10 (Advanced) - 4GB+ GPU
15    4. CelebA (Expert) - 8GB+ GPU
16    ''''')
17 else:
18     print("✅ Dataset variable 'train_dataset' found.")
19
20
21 # Your code to validate GPU memory requirements
22 # Hint: Check torch.cuda.is_available() and use torch.cuda.get_device_properties(0).total_memory
23 # to get available GPU memory, then compare with dataset requirements
24
25 # Enter your code here:
26 import torch # Added import just in case this is a new cell
27
28 # 1. Define memory requirements based on the loaded dataset's variables
29 required_gb = 0
30 dataset_name = "Unknown"
31
32 # Use the variables set in the previous cell to determine requirements
33 if 'IMG_SIZE' in locals() and IMG_SIZE == 28 and IMG_CH == 1:
34     required_gb = 2
35     dataset_name = "MNIST / Fashion-MNIST"
36 elif 'IMG_SIZE' in locals() and IMG_SIZE == 32 and IMG_CH == 3:
37     required_gb = 4
38     dataset_name = "CIFAR-10"
39 # (You could add more elifs here for other datasets like CelebA)
40
41 print(f"ℹ️ Selected dataset ({dataset_name}) requires ~{required_gb} GB of GPU memory.")
42
43 # 2. Check available GPU memory
44 if torch.cuda.is_available():
45     # Get properties of the current GPU (device 0)
46     props = torch.cuda.get_device_properties(0)
47     # Convert total memory from bytes to GiB (1024^3)
48     total_memory_gb = props.total_memory / (1024**3)
49
50     print(f"✅ GPU found: {props.name}")
51     print(f"    - Total Memory: {total_memory_gb:.2f} GB")
52
53     # 3. Compare and validate
54     if total_memory_gb < required_gb:
55         raise ValueError(f"'''")
56         ✘ ERROR: GPU memory insufficient for {dataset_name}!
57         - Required: ~{required_gb} GB
58         - Available: {total_memory_gb:.2f} GB
59         Please restart the runtime, select a smaller dataset (e.g., MNIST),
60         or get a session with a more powerful GPU.
61         ''''")
62     else:
63         print(f"✅ GPU memory is sufficient.")
64
65 else:
66     # Warning if no GPU is found, as training will be very slow
67     print("⚠️ WARNING: No GPU found (torch.cuda.is_available() is False).")
68     print("    Training will run on the CPU, which will be extremely slow.")
69     if required_gb > 2:
70         print(f"    This dataset ({dataset_name}) is not recommended for CPU-only training.")

    ✅ Dataset variable 'train_dataset' found.
    ℹ️ Selected dataset (MNIST / Fashion-MNIST) requires ~2 GB of GPU memory.
    ✅ GPU found: Tesla T4
    - Total Memory: 14.74 GB
    ✅ GPU memory is sufficient.

```

```

1 #Dataset Properties and Data Loaders
2 #Now let's examine our dataset
3 #and set up the data loaders:
4
5 # (Importing necessary libraries)
6 from torch.utils.data import DataLoader, random_split
7 import torch
8

```

```
8 import os
9
10 # Your code to check sample batch properties
11 # Hint: Get a sample batch using next(iter(DataLoader(dataset, batch_size=1)))
12 # Then print information about the dataset shape, type, and value ranges
13
14 # Enter your code here:
15 print("---- 1. Dataset Sample Check ---")
16 # Create a temporary loader to grab one sample
17 temp_loader = DataLoader(train_dataset, batch_size=1, shuffle=True)
18 sample_image, sample_label = next(iter(temp_loader))
19
20 print(f"Sample image batch shape: {sample_image.shape}")
21 print(f"Sample image data type: {sample_image.dtype}")
22 print(f"Sample image value range: Min={sample_image.min():.2f}, Max={sample_image.max():.2f}")
23 print(f"Sample label: {sample_label.item()}")
24 print("Note: A range of -1.0 to 1.0 confirms Normalize((0.5,), (0.5,)) worked.")
25 del temp_loader, sample_image, sample_label # Clean up temp variables
26
27
28 =====
29 # SECTION 3: DATASET SPLITTING AND DATALOADER CONFIGURATION
30 =====
31 # Create train-validation split
32
33 # Your code to create a train-validation split (80% train, 20% validation)
34 # Hint: Use random_split() with appropriate train_size and val_size
35 # Be sure to use a fixed generator for reproducibility
36
37 # Enter your code here:
38 print("\n--- 2. Train/Validation Split ---")
39 dataset_size = len(train_dataset)
40 val_size = int(dataset_size * 0.2) # 20% for validation
41 train_size = dataset_size - val_size # 80% for training
42
43 # Use a fixed generator for reproducible splits
44 generator = torch.Generator().manual_seed(42)
45 train_set, val_set = random_split(
46     train_dataset,
47     [train_size, val_size],
48     generator=generator
49 )
50
51 print(f"Original dataset size: {dataset_size}")
52 print(f"Training set size: {len(train_set)}")
53 print(f"Validation set size: {len(val_set)}")
54
55
56 # Your code to create dataloaders for training and validation
57 # Hint: Use DataLoader with batch_size=BATCH_SIZE, appropriate shuffle settings,
58 # and num_workers based on available CPU cores
59
60 # Enter your code here:
61 print("\n--- 3. DataLoaders Configuration ---")
62 # Use all available CPU cores for loading, or a reasonable number (e.g., 2)
63 # os.cpu_count() is a good default
64 num_workers = min(os.cpu_count(), 8) # Cap at 8 workers to be safe
65 print(f"Using {num_workers} workers for data loading.")
66
67 # BATCH_SIZE was defined in the previous cell (it should be 64 for MNIST)
68
69 train_loader = DataLoader(
70     train_set,
71     batch_size=BATCH_SIZE,
72     shuffle=True, # Shuffle training data
73     num_workers=num_workers,
74     pin_memory=True # Speeds up data transfer to GPU
75 )
76
77 val_loader = DataLoader(
78     val_set,
79     batch_size=BATCH_SIZE,
80     shuffle=False, # No need to shuffle validation data
81     num_workers=num_workers,
82     pin_memory=True
83 )
84
85 print("✅ Successfully created train loader and val loader.")
```

```

--- 1. Dataset Sample Check ---
Sample image batch shape: torch.Size([1, 1, 28, 28])
Sample image data type: torch.float32
Sample image value range: Min=-1.00, Max=0.99
Sample label: 1
Note: A range of -1.0 to 1.0 confirms Normalize((0.5,), (0.5,)) worked.

--- 2. Train/Validation Split ---
Original dataset size: 60000
Training set size: 48000
Validation set size: 12000

--- 3. DataLoaders Configuration ---
Using 2 workers for data loading.
✓ Successfully created train_loader and val_loader.

```

## ▼ 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

```

1 # Basic building block that processes images
2 class GELUConvBlock(nn.Module):
3     def __init__(self, in_ch, out_ch, group_size):
4         """
5             Creates a block with convolution, normalization, and activation
6
7             Args:
8                 in_ch (int): Number of input channels
9                 out_ch (int): Number of output channels
10                group_size (int): Number of groups for GroupNorm
11            """
12        super().__init__()
13
14        # Check that group_size is compatible with out_ch
15        if out_ch % group_size != 0:
16            print(f"Warning: out_ch {out_ch} is not divisible by group_size {group_size}")
17            # Adjust group_size to be compatible
18            group_size = min(group_size, out_ch)
19            while out_ch % group_size != 0:
20                group_size -= 1
21            print(f"Adjusted group_size to {group_size}")
22
23        # Your code to create layers for the block
24        # Hint: Use nn.Conv2d, nn.GroupNorm, and nn.GELU activation
25        # Then combine them using nn.Sequential
26
27        # Enter your code here:
28
29    def forward(self, x):
30        # Your code for the forward pass
31        # Hint: Simply pass the input through the model
32
33        # Enter your code here:
34        pass

```

```

1 # (This assumes you have already run: import torch.nn as nn)
2
3 # Basic building block that processes images
4 class GELUConvBlock(nn.Module):
5     def __init__(self, in_ch, out_ch, group_size):
6         """
7             Creates a block with convolution, normalization, and activation
8
9             Args:
10                in_ch (int): Number of input channels
11                out_ch (int): Number of output channels
12                group_size (int): Number of groups for GroupNorm

```

```

13     """
14     super().__init__()
15
16     # Check that group_size is compatible with out_ch
17     if out_ch % group_size != 0:
18         print(f"Warning: out_ch ({out_ch}) is not divisible by group_size ({group_size})")
19         # Adjust group_size to be compatible
20         group_size = min(group_size, out_ch)
21         while out_ch % group_size != 0:
22             group_size -= 1
23         print(f"Adjusted group_size to {group_size}")
24
25     # Your code to create layers for the block
26     # Hint: Use nn.Conv2d, nn.GroupNorm, and nn.GELU activation
27     # Then combine them using nn.Sequential
28
29     # Enter your code here:
30     self.model = nn.Sequential(
31         # 3x3 convolution with padding=1 to keep image size the same
32         nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=1),
33         # Group normalization
34         nn.GroupNorm(group_size, out_ch),
35         # GELU activation
36         nn.GELU()
37     )
38
39     def forward(self, x):
40         # Your code for the forward pass
41         # Hint: Simply pass the input through the model
42
43         # Enter your_code here:
44         return self.model(x)

```

```

1 # Rearranges pixels to downsample the image (2x reduction in spatial dimensions)
2 class RearrangePoolBlock(nn.Module):
3     def __init__(self, in_chs, group_size):
4         """
5             Downsamples the spatial dimensions by 2x while preserving information
6
7             Args:
8                 in_chs (int): Number of input channels
9                 group_size (int): Number of groups for GroupNorm
10            """
11         super().__init__()
12
13         # Your code to create the rearrange operation and convolution
14         # Hint: Use Rearrange from einops.layers.torch to reshape pixels
15         # Then add a GELUConvBlock to process the rearranged tensor
16
17         # Enter your code here:
18
19     def forward(self, x):
20         # Your code for the forward pass
21         # Hint: Apply rearrange to downsample, then apply convolution
22
23         # Enter your code here:
24         pass

```

```

1 # (This assumes you have already run:
2 # from einops.layers.torch import Rearrange
3 # )
4
5 # Rearranges pixels to downsample the image (2x reduction in spatial dimensions)
6 class RearrangePoolBlock(nn.Module):
7     def __init__(self, in_chs, group_size):
8         """
9             Downsamples the spatial dimensions by 2x while preserving information
10
11             Args:
12                 in_chs (int): Number of input channels
13                 group_size (int): Number of groups for GroupNorm
14            """
15         super().__init__()
16
17         # Your code to create the rearrange operation and convolution
18         # Hint: Use Rearrange from einops.layers.torch to reshape pixels

```

```

19     # Then add a GELUConvBlock to process the rearranged tensor
20
21     # Enter your code here:
22
23     # This operation takes 2x2 patches and moves them to the channel dimension
24     # 'b c (h 2) (w 2)' -> 'b (c 4) h w'
25     # This reduces H and W by 2, and increases C by 4
26     self.rearrange = Rearrange('b c (h 2) (w 2) -> b (c 4) h w')
27
28     # The number of input channels for the conv block is now 4 * in_chs
29     # We'll keep the output channels the same for this block
30     new_chs = in_chs * 4
31
32     # We need to make sure the group_size is valid for the new channel count
33     if new_chs % group_size != 0:
34         # Adjust group_size to be a divisor of new_chs
35         valid_group_size = group_size
36         while new_chs % valid_group_size != 0:
37             valid_group_size -= 1
38         print(f"RearrangePoolBlock adjusted group_size from {group_size} to {valid_group_size} for {new_chs}")
39         group_size = valid_group_size
40
41     self.conv_block = GELUConvBlock(new_chs, new_chs, group_size)
42
43 def forward(self, x):
44     # Your code for the forward pass
45     # Hint: Apply rearrange to downsample, then apply convolution
46
47     # Enter your code here:
48     # 1. Downsample by rearrangement
49     x = self.rearrange(x)
50     # 2. Process with convolution
51     x = self.conv_block(x)
52     return x

```

```

1 # (This assumes 'torch', 'torch.nn as nn', and 'GELUConvBlock' are defined)
2
3 #Now let's implement the upsampling block for our U-Net architecture:
4 class UpBlock(nn.Module):
5     """
6         Upsampling block for decoding path in U-Net architecture.
7
8         This block:
9             1. Takes features from the decoding path and corresponding skip connection
10            2. Concatenates them along the channel dimension
11            3. Upsamples spatial dimensions by 2x using transposed convolution
12            4. Processes features through multiple convolutional blocks
13
14         Args:
15             in_chs (int): Number of input channels from the previous layer
16             out_chs (int): Number of output channels
17             group_size (int): Number of groups for GroupNorm
18     """
19     def __init__(self, in_chs, out_chs, group_size):
20         super().__init__()
21
22         # Your code to create the upsampling operation
23         # Hint: Use nn.ConvTranspose2d with kernel_size=2 and stride=2
24         # This layer upsamples the input 'x' from [B, in_chs, H, W]
25         # to match the skip connection's size: [B, in_chs, 2H, 2W]
26         # Enter your code here:
27         self.up = nn.ConvTranspose2d(in_chs, in_chs, kernel_size=2, stride=2)
28
29
30         # Your code to create the convolutional blocks
31         # Hint: Use multiple GELUConvBlocks in sequence
32         # After concatenation, channels will be (in_chs + in_chs) = 2 * in_chs
33         # These blocks process the concatenated features and output 'out_chs'
34         # Enter your code here:
35         self.convs = nn.Sequential(
36             # First block takes concatenated features and outputs 'out_chs'
37             GELUConvBlock(2 * in_chs, out_chs, group_size),
38             # Second block refines the features
39             GELUConvBlock(out_chs, out_chs, group_size)
40         )
41

```

```

42     # Log the configuration for debugging
43     print(f"Created UpBlock: in_chs={in_chs}, out_chs={out_chs}, spatial_increase=2x")
44
45     def forward(self, x, skip):
46         """
47             Forward pass through the UpBlock.
48
49             Args:
50                 x (torch.Tensor): Input tensor from previous layer [B, in_chs, H, W]
51                 skip (torch.Tensor): Skip connection tensor from encoder [B, in_chs, 2H, 2W]
52
53             Returns:
54                 torch.Tensor: Output tensor with shape [B, out_chs, 2H, 2W]
55             """
56             # Your code for the forward pass
57             # Hint: Upsample x, then concatenate with skip, then process
58
59             # Enter your code here:
60
61             # 1. Upsample x to match skip's spatial dimensions
62             x_up = self.up(x) # Shape: [B, in_chs, 2H, 2W]
63
64             # 2. Concatenate along the channel dimension (dim=1)
65             x_cat = torch.cat([x_up, skip], dim=1) # Shape: [B, 2*in_chs, 2H, 2W]
66
67             # 3. Process with convolutional blocks
68             return self.convs(x_cat) # Shape: [B, out_chs, 2H, 2W]

```

```

1 #=====
2 # SECTION 4: DIFFUSION NOISE SCHEDULE
3 #=====
4 # We define the "noise schedule" – how much noise we add at each timestep.
5 # This is a critical part of the diffusion model.
6
7 # Number of steps in the diffusion process
8 n_steps = 100 # This is the 'T' in the UNet
9 beta_start = 1e-4
10 beta_end = 0.02
11
12 # 1. Create the 'beta' schedule (how much noise to add at step t)
13 # We use a linear schedule for simplicity
14 betas = torch.linspace(beta_start, beta_end, n_steps).to(device)
15
16 # 2. Calculate 'alphas', which represent the 'signal rate' (1 – noise)
17 alphas = 1.0 – betas
18
19 # 3. Calculate 'alpha_bar' (cumulative product of alphas)
20 # This tells us the total signal rate at step t
21 alpha_bar = torch.cumprod(alphas, dim=0)
22
23 # 4. Pre-calculate values for the forward 'add_noise' process
24 # These are the terms in the formula:  $x_t = \sqrt{\alpha_{bar}} * x_0 + \sqrt{1 - \alpha_{bar}} * noise$ 
25 sqrt_alpha_bar = torch.sqrt(alpha_bar)
26 sqrt_one_minus_alpha_bar = torch.sqrt(1.0 – alpha_bar)
27
28 # 5. Pre-calculate values for the reverse 'remove_noise' process
29 # These are the terms needed for the DDPM sampling formula
30 sqrt_recip_alpha = torch.sqrt(1.0 / alphas)
31 sqrt_recip_alpha_bar = torch.sqrt(1.0 / alpha_bar)
32 sqrt_recip_m1_alpha_bar = torch.sqrt(1.0 / alpha_bar – 1)
33 posterior_variance = betas * (1.0 – torch.roll(alpha_bar, 1, 0)) / (1.0 – alpha_bar)
34 posterior_variance[0] = betas[0] # Set first value (no previous alpha_bar)
35 posterior_log_variance = torch.log(posterior_variance.clamp(min=1e-20))
36
37 print(f"✅ Noise schedule created with {n_steps} steps.")
38 print(f"    - betas: {betas.shape}")
39 print(f"    - alpha_bar: {alpha_bar.shape}")

```

- ✓ Noise schedule created with 100 steps.
  - betas: torch.Size([100])
  - alpha\_bar: torch.Size([100])

```

1 # (This assumes the following are defined:
2 # import torch
3 # from torch.optim import Adam
4 # model, device, IMG_SIZE, IMG_CH, n_steps, N_CLASSES,
# # + loss function

```

```

3 # train_loader, val_loader, device
4 #
5 from torch.optim import Adam # Added import for the optimizer
6
7 # Create our model and move it to GPU if available
8 model = UNet(
9     T=n_steps,                      # Number of diffusion time steps
10    img_ch=IMG_CH,                  # Number of channels in our images (1 for grayscale, 3 for RGB)
11    img_size=IMG_SIZE,              # Size of input images (28 for MNIST, 32 for CIFAR-10)
12    down_chs=(32, 64, 128),        # Channel dimensions for each downsampling level
13    t_embed_dim=8,                 # Dimension for time step embeddings
14    c_embed_dim=N_CLASSES         # Number of classes for conditioning
15 ).to(device)
16
17 # Print model summary
18 print(f"\n{'='*50}")
19 print(f"MODEL ARCHITECTURE SUMMARY")
20 print(f"{'='*50}")
21 print(f"Input resolution: {IMG_SIZE}x{IMG_SIZE}")
22 print(f"Input channels: {IMG_CH}")
23 print(f"Time steps: {n_steps}")
24 print(f"Condition classes: {N_CLASSES}")
25 print(f"GPU acceleration: {'Yes' if device.type == 'cuda' else 'No'}")
26
27 # Validate model parameters and estimate memory requirements
28 # Hint: Create functions to count parameters and estimate memory usage
29
30 # Enter your code here:
31 def count_parameters(model):
32     """Counts the total number of trainable parameters in a model."""
33     return sum(p.numel() for p in model.parameters() if p.requires_grad)
34
35 total_params = count_parameters(model)
36 print(f"Total Trainable Parameters: {total_params:,} (~{total_params/1e6:.2f} M)")
37
38 if device.type == 'cuda':
39     # Memory already allocated just for the model weights
40     allocated_mb = torch.cuda.memory_allocated(device) / (1024**2)
41     print(f"Model VRAM (weights only): {allocated_mb:.2f} MB")
42     print("Note: Total VRAM usage during training will be much higher due to gradients,")
43     print("      optimizer states (Adam), and batch activations.")
44
45
46
47 # Your code to verify data ranges and integrity
48 # Hint: Create functions to check data ranges in training and validation data
49
50 # Enter your code here:
51 def check_data_loader(loader, name):
52     """Grabs one batch and prints its properties to check integrity."""
53     print(f"\n--- Checking {name} ---")
54     try:
55         # Get one batch and move it to the CPU for checking
56         images, labels = next(iter(loader))
57         images, labels = images.cpu(), labels.cpu()
58
59         print(f"  Image batch shape: {images.shape}")
60         print(f"  Image data type: {images.dtype}")
61         print(f"  Image min/max/mean: {images.min():.2f} / {images.max():.2f} / {images.mean():.2f}")
62         print(f"  Label batch shape: {labels.shape}")
63         print(f"  Label data type: {labels.dtype}")
64         print(f"  Label min/max: {labels.min()} / {labels.max()}")
65         print(f"  Image has NaNs: {torch.isnan(images).any()}")
66         print(f"  Image has Infs: {torch.isinf(images).any()}")
67
68     except Exception as e:
69         print(f"  Error checking {name}: {e}")
70
71 print(f"\n{'='*50}")
72 print(f"DATA LOADER INTEGRITY CHECK")
73 print(f"{'='*50}")
74 check_data_loader(train_loader, "Training Loader")
75 check_data_loader(val_loader, "Validation Loader")
76 print("\nCheck: Image min/max should be approx. [-1.0, 1.0].")
77 print("Check: Label min/max should be [0, 9] for MNIST/FashionMNIST.")
78
79 # Set up the optimizer with parameters tuned for diffusion models
80 # Note: Lower learning rates tend to work better for diffusion models
81 initial_lr = 0.001 # Starting learning rate

```

```

83 weight_decay = 1e-5 # L2 regularization to prevent overfitting
84
85 optimizer = Adam(
86     model.parameters(),
87     lr=initial_lr,
88     weight_decay=weight_decay
89 )
90
91 # Learning rate scheduler to reduce LR when validation loss plateaus
92 # This helps fine-tune the model toward the end of training
93 scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
94     optimizer,
95     mode='min',           # Reduce LR when monitored value stops decreasing
96     factor=0.5,          # Multiply LR by this factor
97     patience=5,          # Number of epochs with no improvement after which LR will be reduced
98     # verbose=True,       # <-- THIS LINE WAS REMOVED TO FIX THE TypeError
99     min_lr=1e-6          # Lower bound on the learning rate
100 )
101
102 print("\n✅ Optimizer (Adam) and Scheduler (ReduceLROnPlateau) are set up.")
103
104 # STUDENT EXPERIMENT:
105 # Try different channel configurations and see how they affect:
106 # 1. Model size (parameter count)
107 # 2. Training time
108 # 3. Generated image quality
109 #
110 # Suggestions:
111 # - Smaller: down_chs=(16, 32, 64)
112 # - Larger: down_chs=(64, 128, 256, 512)

```

---

**MODEL ARCHITECTURE SUMMARY**


---

Input resolution: 28x28  
 Input channels: 1  
 Time steps: 100  
 Condition classes: 10  
 GPU acceleration: Yes  
 Total Trainable Parameters: 3,230,412 (~3.23 M)  
 Model VRAM (weights only): 12.33 MB  
 Note: Total VRAM usage during training will be much higher due to gradients,  
 optimizer states (Adam), and batch activations.

---

**DATA LOADER INTEGRITY CHECK**


---

--- Checking Training Loader ---  
 Image batch shape: torch.Size([64, 1, 28, 28])  
 Image data type: torch.float32  
 Image min/max/mean: -1.00 / 1.00 / -0.74  
 Label batch shape: torch.Size([64])  
 Label data type: torch.int64  
 Label min/max: 0 / 9  
 Image has NaNs: False  
 Image has Infs: False

--- Checking Validation Loader ---  
 Image batch shape: torch.Size([64, 1, 28, 28])  
 Image data type: torch.float32  
 Image min/max/mean: -1.00 / 1.00 / -0.73  
 Label batch shape: torch.Size([64])  
 Label data type: torch.int64  
 Label min/max: 0 / 9  
 Image has NaNs: False  
 Image has Infs: False

Check: Image min/max should be approx. [-1.0, 1.0].  
 Check: Label min/max should be [0, 9] for MNIST/FashionMNIST.

✅ Optimizer (Adam) and Scheduler (ReduceLROnPlateau) are set up.

```

1 import torch
2 import torch.nn as nn
3 from einops.layers.torch import Rearrange
4 import torch.nn.functional as F
5
6 # 1. HELPER CLASS: GELUConvBlock

```

```

7 class GELUConvBlock(nn.Module):
8     def __init__(self, in_ch, out_ch, group_size):
9         super().__init__()
10        # Fix group_size if not divisible
11        if out_ch % group_size != 0:
12            valid_group_size = group_size
13            while out_ch % valid_group_size != 0 and valid_group_size > 1:
14                valid_group_size -= 1
15            if out_ch % valid_group_size != 0: # Failsafe
16                valid_group_size = 1
17            group_size = valid_group_size
18
19        self.model = nn.Sequential(
20            nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=1),
21            nn.GroupNorm(group_size, out_ch),
22            nn.GELU()
23        )
24    def forward(self, x):
25        return self.model(x)
26
27 # 2. HELPER CLASS: RearrangePoolBlock
28 class RearrangePoolBlock(nn.Module):
29     def __init__(self, in_chs, group_size):
30         super().__init__()
31         # Fix for EinopsError: Use named parameters p1=2, p2=2
32         self.rearrange = Rearrange('b c (h p1) (w p2) -> b (c p1 p2) h w', p1=2, p2=2)
33         new_chs = in_chs * 4
34
35         # Fix group_size for new channel count
36         if new_chs % group_size != 0:
37             valid_group_size = group_size
38             while new_chs % valid_group_size != 0 and valid_group_size > 1:
39                 valid_group_size -= 1
40             if new_chs % valid_group_size != 0: # Failsafe
41                 valid_group_size = new_chs
42             group_size = valid_group_size
43
44         self.conv_block = GELUConvBlock(new_chs, new_chs, group_size)
45    def forward(self, x):
46        x = self.rearrange(x)
47        x = self.conv_block(x)
48        return x
49
50 # 3. HELPER CLASS: DownBlock
51 class DownBlock(nn.Module):
52     def __init__(self, in_chs, out_chs, group_size):
53         super().__init__()
54         layers = [
55             GELUConvBlock(in_chs, out_chs, group_size),
56             GELUConvBlock(out_chs, out_chs, group_size),
57             RearrangePoolBlock(out_chs, group_size)
58         ]
59         self.model = nn.Sequential(*layers)
60    def forward(self, x):
61        return self.model(x)
62
63 # 4. HELPER CLASS: UpBlock
64 class UpBlock(nn.Module):
65     def __init__(self, in_chs, out_chs, group_size):
66         super().__init__()
67         self.up = nn.ConvTranspose2d(in_chs, in_chs, kernel_size=2, stride=2)
68         self.conv = nn.Sequential(
69             GELUConvBlock(2 * in_chs, out_chs, group_size),
70             GELUConvBlock(out_chs, out_chs, group_size)
71         )
72    def forward(self, x, skip):
73        x_up = self.up(x)
74        x_cat = torch.cat([x_up, skip], dim=1)
75        return self.conv(x_cat)
76
77 # 5. MAIN UNET CLASS (THIS IS THE MISSING ONE)
78 class UNet(nn.Module):
79     def __init__(self, T, img_ch, img_size, down_chs, t_embed_dim, c_embed_dim):
80         super().__init__()
81         GS = 8 # Default Group Size
82         self.down_chs = down_chs
83         self.t_embed_dim = t_embed_dim

```

```

84     self.c_embed_dim = c_embed_dim
85
86     # Time embedding
87     self.time_embed = nn.Sequential(
88         nn.Embedding(T, t_embed_dim),
89         nn.Linear(t_embed_dim, t_embed_dim),
90         nn.GELU()
91     )
92
93     # Class embedding (assumes N_CLASSES is globally defined)
94     self.class_embed = nn.Embedding(N_CLASSES, c_embed_dim)
95
96     # Initial convolution
97     self.init_conv = GELUConvBlock(img_ch, down_chs[0], GS)
98
99     # Downsampling path
100    self.downs = nn.ModuleList()
101    for i in range(len(down_chs) - 1):
102        self.downs.append(
103            DownBlock(down_chs[i], down_chs[i+1], GS)
104        )
105
106    # Middle

```

```

1 import torch
2 import torch.nn as nn
3 from einops.layers.torch import Rearrange
4 import torch.nn.functional as F
5
6 # 1. HELPER CLASS: GELUConvBlock
7 class GELUConvBlock(nn.Module):
8     def __init__(self, in_ch, out_ch, group_size):
9         super().__init__()
10        # Fix group_size if not divisible
11        if out_ch % group_size != 0:
12            valid_group_size = group_size
13            while out_ch % valid_group_size != 0 and valid_group_size > 1:
14                valid_group_size -= 1
15            if out_ch % valid_group_size != 0: # Failsafe
16                valid_group_size = 1
17            group_size = valid_group_size
18
19        self.model = nn.Sequential(
20            nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=1),
21            nn.GroupNorm(group_size, out_ch),
22            nn.GELU()
23        )
24    def forward(self, x):
25        return self.model(x)
26
27 # 2. HELPER CLASS: RearrangePoolBlock
28 class RearrangePoolBlock(nn.Module):
29     def __init__(self, in_chs, group_size):
30         super().__init__()
31         # Fix for EinopsError: Use named parameters p1=2, p2=2
32         self.rearrange = Rearrange('b c (h p1) (w p2) -> b (c p1 p2) h w', p1=2, p2=2)
33         new_chs = in_chs * 4
34
35         # Fix group_size for new channel count
36         if new_chs % group_size != 0:
37             valid_group_size = group_size
38             while new_chs % valid_group_size != 0 and valid_group_size > 1:
39                 valid_group_size -= 1
40             if new_chs % valid_group_size != 0: # Failsafe
41                 valid_group_size = new_chs
42             group_size = valid_group_size
43
44         self.conv_block = GELUConvBlock(new_chs, new_chs, group_size)
45    def forward(self, x):
46        x = self.rearrange(x)
47        x = self.conv_block(x)
48        return x
49
50 # 3. HELPER CLASS: DownBlock
51 class DownBlock(nn.Module):
52     def __init__(self, in_chs, out_chs, group_size):
53         super().__init__()

```

```

54     layers = [
55         GELUConvBlock(in_chs, out_chs, group_size),
56         GELUConvBlock(out_chs, out_chs, group_size),
57         RearrangePoolBlock(out_chs, group_size)
58     ]
59     self.model = nn.Sequential(*layers)
60   def forward(self, x):
61     return self.model(x)
62
63 # 4. HELPER CLASS: UpBlock
64 class UpBlock(nn.Module):
65   def __init__(self, in_chs, out_chs, group_size):
66     super().__init__()
67     self.up = nn.ConvTranspose2d(in_chs, in_chs, kernel_size=2, stride=2)
68     self.conv = nn.Sequential(
69         GELUConvBlock(2 * in_chs, out_chs, group_size),
70         GELUConvBlock(out_chs, out_chs, group_size)
71     )
72   def forward(self, x, skip):
73     x_up = self.up(x)
74     x_cat = torch.cat([x_up, skip], dim=1)
75     return self.conv(x_cat)
76
77 # 5. MAIN UNET CLASS (THIS IS THE MISSING ONE)
78 class UNet(nn.Module):
79   def __init__(self, T, img_ch, img_size, down_chs, t_embed_dim, c_embed_dim):
80     super().__init__()
81     GS = 8 # Default Group Size
82     self.down_chs = down_chs
83     self.t_embed_dim = t_embed_dim
84     self.c_embed_dim = c_embed_dim
85
86     # Time embedding
87     self.time_embed = nn.Sequential(
88         nn.Embedding(T, t_embed_dim),
89         nn.Linear(t_embed_dim, t_embed_dim),
90         nn.GELU()
91     )
92
93     # Class embedding (assumes N_CLASSES is globally defined)
94     self.class_embed = nn.Embedding(N_CLASSES, c_embed_dim)
95
96     # Initial convolution
97     self.init_conv = GELUConvBlock(img_ch, down_chs[0], GS)
98
99     # Downsampling path
100    self.downs = nn.ModuleList()
101    for i in range(len(down_chs) - 1):
102      self.downs.append(
103          DownBlock(down_chs[i], down_chs[i+1], GS)
104      )
105
106    # Middle blocks
107    self.mids = nn.Sequential(
108        GELUConvBlock(down_chs[-1], down_chs[-1], GS),
109        GELUConvBlock(down_chs[-1], down_chs[-1], GS)
110    )
111    self.mid_t_proj = nn.Linear(t_embed_dim, down_chs[-1])
112    self.mid_c_proj = nn.Linear(c_embed_dim, down_chs[-1])
113
114    # Upsampling path (Fixed IndexError)
115    self.ups = nn.ModuleList()
116    for i in range(len(down_chs)-1, 0, -1):
117      self.ups.append(
118          UpBlock(down_chs[i], down_chs[i-1], GS)
119      )
120
121    # Final convolution
122    self.final_conv = nn.Conv2d(down_chs[0], img_ch, kernel_size=1)
123    print(f"✅ Created UNet with {len(down_chs)} scale levels")
124
125  def forward(self, x, t, c, c_mask):
126    """
127      Forward pass through the UNet.
128    """
129    t_embed = self.time_embed(t)
130    c_embed = self.class_embed(c)
131

```

```

131     c_embed = c_embed * c_mask # Apply mask
132     x = self.init_conv(x)
133
134     skips = []
135     for down_block in self.downs:
136         skips.append(x)
137         x = down_block(x)
138
139     x = self.mids(x)
140     b, c_dim, h_dim, w_dim = x.shape
141
142     t_proj = self.mid_t_proj(t_embed).view(b, c_dim, 1, 1)
143     c_proj = self.mid_c_proj(c_embed).view(b, c_dim, 1, 1)
144     x = x + t_proj + c_proj
145
146     # Fixed forward pass logic
147     for up_block in self.ups:
148         skip = skips.pop()
149         x = up_block(x, skip)
150
151     return self.final_conv(x)
152
153 print("✅ All model classes (UNet and helpers) are defined.")

```

All model classes (UNet and helpers) are defined.

```

1 # (This assumes the following are defined:
2 # import torch
3 # from torch.optim import Adam
4 # model, device, IMG_SIZE, IMG_CH, n_steps, N_CLASSES,
5 # train_loader, val_loader
6 #
7 from torch.optim import Adam # Added import for the optimizer
8
9 # Create our model and move it to GPU if available
10 # NOTE: Ensure you have run the cell defining the corrected UNet class
11 # (which uses Conv2d for downsampling instead of RearrangePoolBlock)
12 model = UNet(
13     T=n_steps,                      # Number of diffusion time steps
14     img_ch=IMG_CH,                  # Number of channels in our images (1 for grayscale, 3 for RGB)
15     img_size=IMG_SIZE,              # Size of input images (28 for MNIST, 32 for CIFAR-10)
16     down_chs=(32, 64, 128),        # Channel dimensions for each downsampling level
17     t_embed_dim=8,                 # Dimension for time step embeddings
18     c_embed_dim=N_CLASSES          # Number of classes for conditioning
19 ).to(device)
20
21 # Print model summary
22 print(f"\n{'='*50}")
23 print(f"MODEL ARCHITECTURE SUMMARY")
24 print(f"{'='*50}")
25 print(f"Input resolution: {IMG_SIZE}x{IMG_SIZE}")
26 print(f"Input channels: {IMG_CH}")
27 print(f"Time steps: {n_steps}")
28 print(f"Condition classes: {N_CLASSES}")
29 print(f"GPU acceleration: {'Yes' if device.type == 'cuda' else 'No'}")
30
31 # Validate model parameters and estimate memory requirements
32 # Hint: Create functions to count parameters and estimate memory usage
33
34 # Enter your code here:
35 def count_parameters(model):
36     """Counts the total number of trainable parameters in a model."""
37     return sum(p.numel() for p in model.parameters() if p.requires_grad)
38
39 total_params = count_parameters(model)
40 print(f"Total Trainable Parameters: {total_params:,} (~{total_params/1e6:.2f} M)") # Completed the f-string
41
42 if device.type == 'cuda':
43     # Memory already allocated just for the model weights
44     allocated_mb = torch.cuda.memory_allocated(device) / (1024**2)
45     print(f"Model VRAM (weights only): {allocated_mb:.2f} MB")
46     print("Note: Total VRAM usage during training will be much higher due to gradients,")
47     print("      optimizer states (Adam), and batch activations.")
48
49
50 # Your code to verify data ranges and integrity
51 # Hint: Create functions to check data ranges in training and validation data
--_

```

```

52
53 # Enter your code here:
54 def check_data_loader(loader, name):
55     """Grabs one batch and prints its properties to check integrity."""
56     print(f"\n--- Checking {name} ---")
57     try:
58         # Get one batch and move it to the CPU for checking
59         images, labels = next(iter(loader))
60         images, labels = images.cpu(), labels.cpu()
61
62         print(f"  Image batch shape: {images.shape}")
63         print(f"  Image data type:  {images.dtype}")
64         print(f"  Image min/max/mean: {images.min():.2f} / {images.max():.2f} / {images.mean():.2f}")
65         print(f"  Label batch shape: {labels.shape}")
66         print(f"  Label data type:  {labels.dtype}")
67         print(f"  Label min/max:    {labels.min()} / {labels.max()}")
68         print(f"  Image has NaNs:   {torch.isnan(images).any()}")
69         print(f"  Image has Infs:   {torch.isinf(images).any()}")
70     except Exception as e:
71         print(f"  Error checking {name}: {e}")
72
73 print("\n{'='*50}")
74 print("DATA LOADER INTEGRITY CHECK")
75 print("{'='*50}")
76 check_data_loader(train_loader, "Training Loader")
77 check_data_loader(val_loader, "Validation Loader")
78 print("\nCheck: Image min/max should be approx. [-1.0, 1.0].")
79 print("Check: Label min/max should be [0, 9] for MNIST/FashionMNIST.")
80
81
82 # Set up the optimizer with parameters tuned for diffusion models
83 # Note: Lower learning rates tend to work better for diffusion models
84 initial_lr = 0.001 # Starting learning rate
85 weight_decay = 1e-5 # L2 regularization to prevent overfitting
86
87 optimizer = Adam(
88     model.parameters(),
89     lr=initial_lr,
90     weight_decay=weight_decay
91 )
92
93 # Learning rate scheduler to reduce LR when validation loss plateaus
94 # This helps fine-tune the model toward the end of training
95 scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
96     optimizer,
97     mode='min',           # Reduce LR when monitored value stops decreasing
98     factor=0.5,          # Multiply LR by this factor
99     patience=5,          # Number of epochs with no improvement after which LR will be reduced
100    # verbose=True,       # Removed deprecated argument
101    min_lr=1e-6          # Lower bound on the learning rate
102 )
103
104 print("\n✅ Optimizer (Adam) and Scheduler (ReduceLROnPlateau) are set up.")
105
106 # STUDENT EXPERIMENT:
107 # Try different channel configurations and see how they affect:
108 # 1. Model size (parameter count)
109 # 2. Training time
110 # 3. Generated image quality
111 #
112 # Suggestions:
113 # - Smaller: down_chs=(16, 32, 64)
114 # - Larger: down_chs=(64, 128, 256, 512)

```

Created UNet with 3 scale levels

```
=====
MODEL ARCHITECTURE SUMMARY
=====
Input resolution: 28x28
Input channels: 1
Time steps: 100
Condition classes: 10
GPU acceleration: Yes
Total Trainable Parameters: 3,841,773 (~3.84 M)
Model VRAM (weights only): 27.00 MB
Note: Total VRAM usage during training will be much higher due to gradients,
      optimizer states (Adam), and batch activations.
```

```
=====
DATA LOADER INTEGRITY CHECK
=====
```

```
--- Checking Training Loader ---
Image batch shape: torch.Size([64, 1, 28, 28])
Image data type: torch.float32
Image min/max/mean: -1.00 / 1.00 / -0.74
Label batch shape: torch.Size([64])
Label data type: torch.int64
Label min/max: 0 / 9
Image has NaNs: False
Image has Infs: False
```

```
--- Checking Validation Loader ---
Image batch shape: torch.Size([64, 1, 28, 28])
Image data type: torch.float32
Image min/max/mean: -1.00 / 1.00 / -0.73
Label batch shape: torch.Size([64])
Label data type: torch.int64
Label min/max: 0 / 9
Image has NaNs: False
Image has Infs: False
```

Check: Image min/max should be approx. [-1.0, 1.0].  
 Check: Label min/max should be [0, 9] for MNIST/FashionMNIST.

Optimizer (Adam) and Scheduler (ReduceLROnPlateau) are set up.

```
1 # (This assumes you have already run:
2 # import torch
3 # import torch.nn as nn
4 #
5
6 #Now let's implement the upsampling block for our U-Net architecture:
7 class UpBlock(nn.Module):
8     """
9         Upsampling block for decoding path in U-Net architecture.
10
11    This block:
12        1. Takes features from the decoding path and corresponding skip connection
13        2. Concatenates them along the channel dimension
14        3. Upsamples spatial dimensions by 2x using transposed convolution
15        4. Processes features through multiple convolutional blocks
16
17    Args:
18        in_chs (int): Number of input channels from the previous layer
19        out_chs (int): Number of output channels
20        group_size (int): Number of groups for GroupNorm
21    """
22    def __init__(self, in_chs, out_chs, group_size):
23        super().__init__()
24
25        # Your code to create the upsampling operation
26        # Hint: Use nn.ConvTranspose2d with kernel_size=2 and stride=2
27        # Note that the input channels will be 2 * in_chs due to concatenation
28
29        # This layer upsamples the input 'x' from the layer below.
30        # It takes 'in_chs' and produces 'out_chs' to match the skip connection.
31        # (Note: The prompt's docstring/hints are slightly confusing.
32        # A standard U-Net upsamples 'x' from [B, in_chs, H, W] to [B, out_chs, 2H, 2W],
33        # then concatenates with 'skip' [B, out_chs, 2H, 2W]
34        # making the input to the convs [B, 2*out_chs, 2H, 2W].
35        # We will follow the prompt's hint literally,
36        # assuming the 'skip' tensor has 'in_chs' channels.)
37
38        # Based on the hint/docstring, we upsample in_chs -> in_chs
39        self.up = nn.ConvTranspose2d(in_chs, in_chs, kernel_size=2, stride=2)
40
41        # Enter your code here: (This part is handled by self.up above)
42
43
44        # Your code to create the convolutional blocks
45        # Hint: Use multiple GELUConvBlocks in sequence
46
47        # Input to convs will be 2 * in_chs (from upsampled 'x' + 'skip')
48        # Output should be 'out_chs'
49        # We use two conv blocks to process the concatenated features
50        self.conv = nn.Sequential(
```

```

51         # First block reduces channels from 2*in_chs to out_chs
52         GELUConvBlock(2 * in_chs, out_chs, group_size),
53         # Second block refines the features at the out_chs dimension
54         GELUConvBlock(out_chs, out_chs, group_size)
55     )
56
57     # Enter your code here: (This part is handled by self.conv above)
58
59     # Log the configuration for debugging
60     print(f"Created UpBlock: in_chs={in_chs}, out_chs={out_chs}, spatial_increase=2x")
61
62 def forward(self, x, skip):
63     """
64     Forward pass through the UpBlock.
65
66     Args:
67         x (torch.Tensor): Input tensor from previous layer [B, in_chs, H, W]
68         skip (torch.Tensor): Skip connection tensor from encoder [B, in_chs, 2H, 2W]
69
70     Returns:
71         torch.Tensor: Output tensor with shape [B, out_chs, 2H, 2W]
72     """
73
74     # Your code for the forward pass
75     # Hint: Concatenate x and skip, then upsample and process
76
77     # Enter your code here:
78
79     # 1. Upsample x to match the spatial dimensions of skip
80     x_up = self.up(x) # Shape: [B, in_chs, 2H, 2W]
81
82     # 2. Concatenate along the channel dimension (dim=1)
83     x_cat = torch.cat([x_up, skip], dim=1) # Shape: [B, 2*in_chs, 2H, 2W]
84
85     # 3. Process with convolutional blocks
86     return self.conv(x_cat) # Shape: [B, out_chs, 2H, 2W]

```

```

1 # Here we implement the time embedding block for our U-Net architecture:
2 # Helps the model understand time steps in diffusion process
3 class SinusoidalPositionEmbedBlock(nn.Module):
4     """
5     Creates sinusoidal embeddings for time steps in diffusion process.
6
7     This embedding scheme is adapted from the Transformer architecture and
8     provides a unique representation for each time step that preserves
9     relative distance information.
10
11    Args:
12        dim (int): Embedding dimension
13    """
14    def __init__(self, dim):
15        super().__init__()
16        self.dim = dim
17
18    def forward(self, time):
19        """
20        Computes sinusoidal embeddings for given time steps.
21
22        Args:
23            time (torch.Tensor): Time steps tensor of shape [batch_size]
24
25        Returns:
26            torch.Tensor: Time embeddings of shape [batch_size, dim]
27        """
28        device = time.device
29        half_dim = self.dim // 2
30        embeddings = torch.log(torch.tensor(10000.0, device=device)) / (half_dim - 1)
31        embeddings = torch.exp(torch.arange(half_dim, device=device) * -embeddings)
32        embeddings = time[:, None] * embeddings[None, :]
33        embeddings = torch.cat((embeddings.sin(), embeddings.cos()), dim=-1)
34        return embeddings
35
36

```

```

1 # Helps the model understand which number/image to draw (class conditioning)
2 class EmbedBlock(nn.Module):
3     """

```

```

4     Creates embeddings for class conditioning in diffusion models.
5
6     This module transforms a one-hot or index representation of a class
7     into a rich embedding that can be added to feature maps.
8
9     Args:
10        input_dim (int): Input dimension (typically number of classes)
11        emb_dim (int): Output embedding dimension
12    """
13    def __init__(self, input_dim, emb_dim):
14        super(EmbedBlock, self).__init__()
15        self.input_dim = input_dim
16
17        # Your code to create the embedding layers
18        # Hint: Use nn.Linear layers with a GELU activation, followed by
19        # nn.Unflatten to reshape for broadcasting with feature maps
20
21        # Enter your code here:
22
23
24
25    def forward(self, x):
26    """
27        Computes class embeddings for the given class indices.
28
29        Args:
30            x (torch.Tensor): Class indices or one-hot encodings [batch_size, input_dim]
31
32        Returns:
33            torch.Tensor: Class embeddings of shape [batch_size, emb_dim, 1, 1]
34                (ready to be added to feature maps)
35    """
36    x = x.view(-1, self.input_dim)
37    return self.model(x)
38
39

```

```

1 # Main U-Net model that puts everything together
2 class UNet(nn.Module):
3     """
4         U-Net architecture for diffusion models with time and class conditioning.
5
6         This architecture follows the standard U-Net design with:
7         1. Downsampling path that reduces spatial dimensions
8         2. Middle processing blocks
9         3. Upsampling path that reconstructs spatial dimensions
10        4. Skip connections between symmetric layers
11
12        The model is conditioned on:
13        - Time step (where we are in the diffusion process)
14        - Class labels (what we want to generate)
15
16        Args:
17            T (int): Number of diffusion time steps
18            img_ch (int): Number of image channels
19            img_size (int): Size of input images
20            down_chs (list): Channel dimensions for each level of U-Net
21            t_embed_dim (int): Dimension for time embeddings
22            c_embed_dim (int): Dimension for class embeddings
23        """
24    def __init__(self, T, img_ch, img_size, down_chs, t_embed_dim, c_embed_dim):
25        super().__init__()
26
27        # Your code to create the time embedding
28        # Hint: Use SinusoidalPositionEmbedBlock, nn.Linear, and nn.GELU in sequence
29
30        # Enter your code here:
31
32        # Your code to create the class embedding
33        # Hint: Use the EmbedBlock class you defined earlier
34
35        # Enter your code here:
36
37        # Your code to create the initial convolution
38        # Hint: Use GELUConvBlock to process the input image
39

```

```
40     # Enter your code here:
41
42     # Your code to create the downsampling path
43     # Hint: Use nn.ModuleList with DownBlock for each level
44
45     # Enter your code here:
46
47     # Your code to create the middle blocks
48     # Hint: Use GELUConvBlock twice to process features at lowest resolution
49
50     # Enter your code here:
51
52     # Your code to create the upsampling path
53     # Hint: Use nn.ModuleList with UpBlock for each level (in reverse order)
54
55     # Enter your code here:
56
57     # Your code to create the final convolution
58     # Hint: Use nn.Conv2d to project back to the original image channels
59
60     # Enter your code here:
61
62     print(f"Created UNet with {len(down_chs)} scale levels")
63     print(f"Channel dimensions: {down_chs}")
64
65 def forward(self, x, t, c, c_mask):
66     """
67         Forward pass through the UNet.
68
69         Args:
70             x (torch.Tensor): Input noisy image [B, img_ch, H, W]
71             t (torch.Tensor): Diffusion time steps [B]
72             c (torch.Tensor): Class labels [B, c_embed_dim]
73             c_mask (torch.Tensor): Mask for conditional generation [B, 1]
74
75         Returns:
76             torch.Tensor: Predicted noise in the input image [B, img_ch, H, W]
77             """
78
79         # Your code for the time embedding
80         # Hint: Process the time steps through the time embedding module
81
82         # Enter your code here:
83
84         # Your code for the class embedding
85         # Hint: Process the class labels through the class embedding module
86
87         # Enter your code here:
88
89         # Your code for the initial feature extraction
90         # Hint: Apply initial convolution to the input
91
92         # Enter your code here:
93
94         # Your code for the downsampling path and skip connections
95         # Hint: Process the features through each downsampling block
96         # and store the outputs for skip connections
97
98         # Enter your code here:
99
100        # Your code for the middle processing and conditioning
101        # Hint: Process features through middle blocks, then add time and class embeddings
102
103        # Enter your code here:
104
105        # Your code for the upsampling path with skip connections
106        # Hint: Process features through each upsampling block,
107        # combining with corresponding skip connections
108
109        # Enter your code here:
110
111        # Your code for the final projection
112        # Hint: Apply the final convolution to get output in image space
113
114        # Enter your code here:
115        pass
```

```

1 # (This assumes 'torch', 'torch.nn' as nn', 'N_CLASSES',
2 # 'GELUConvBlock', 'DownBlock', and 'UpBlock' are defined)
3
4 # Main U-Net model that puts everything together
5 class UNet(nn.Module):
6     """
7         U-Net architecture for diffusion models with time and class conditioning.
8         ...
9         """
10    def __init__(self, T, img_ch, img_size, down_chs, t_embed_dim, c_embed_dim):
11        super().__init__()
12
13        # Define a standard group size for GroupNorm
14        GS = 8
15        self.down_chs = down_chs
16
17        # Time embedding
18        self.time_embed = nn.Sequential(
19            nn.Embedding(T, t_embed_dim),
20            nn.Linear(t_embed_dim, t_embed_dim),
21            nn.GELU()
22        )
23
24        # Class embedding (assumes N_CLASSES is globally defined)
25        self.class_embed = nn.Embedding(N_CLASSES, c_embed_dim)
26
27        # Initial convolution
28        self.init_conv = GELUConvBlock(img_ch, down_chs[0], GS)
29
30        # Downsampling path
31        self.downs = nn.ModuleList()
32        for i in range(len(down_chs) - 1):
33            self.downs.append(DownBlock(down_chs[i], down_chs[i+1], GS))
34
35        # Middle blocks
36        self.mids = nn.Sequential(
37            GELUConvBlock(down_chs[-1], down_chs[-1], GS), # Fixed this line
38            GELUConvBlock(down_chs[-1], down_chs[-1], GS) # Fixed this line
39        )
40
41        # Linear layers to project embeddings
42        self.mid_t_proj = nn.Linear(t_embed_dim, down_chs[-1])
43        self.mid_c_proj = nn.Linear(c_embed_dim, down_chs[-1])
44
45        # Upsampling path
46        self.ups = nn.ModuleList()
47        for i in range(len(down_chs)-1, 0, -1):
48            self.ups.append(UpBlock(down_chs[i], down_chs[i-1], GS))
49
50        # Final convolution
51        self.final_conv = nn.Conv2d(down_chs[0], img_ch, kernel_size=1)
52
53        print(f"✅ Created UNet with {len(down_chs)} scale levels")
54        print(f"    Channel dimensions: {down_chs}")
55
56    def forward(self, x, t, c, c_mask):
57        """
58            Forward pass through the UNet.
59            """
60
61        # Time embedding
62        t_embed = self.time_embed(t)
63
64        # Class embedding
65        c_embed = self.class_embed(c)
66        c_embed = c_embed * c_mask
67
68        # Initial feature extraction
69        x = self.init_conv(x)
70
71        # Downsampling path
72        skips = []
73        for down_block in self.downs:
74            skips.append(x)
75            x = down_block(x)
76
77        # Middle processing

```

```

78     x = self.mids(x)
79     b, c_dim, h_dim, w_dim = x.shape
80
81     # Project embeddings and reshape
82     t_proj = self.mid_t_proj(t_embed).view(b, c_dim, 1, 1)
83     c_proj = self.mid_c_proj(c_embed).view(b, c_dim, 1, 1)
84
85     # Add conditioning
86     x = x + t_proj + c_proj
87
88     # Upsampling path
89     for up_block in self.ups:
90         skip = skips.pop()
91         x = up_block(x, skip)
92
93     # Final projection
94     return self.final_conv(x)

```

```

1 # (This assumes 'torch', 'torch.nn' as nn', 'N_CLASSES',
2 # 'GELUConvBlock', 'DownBlock', and 'UpBlock' are defined)
3
4 # Main U-Net model that puts everything together
5 class UNet(nn.Module):
6     """
7         U-Net architecture for diffusion models with time and class conditioning.
8         ... (docstring) ...
9     """
10    def __init__(self, T, img_ch, img_size, down_chs, t_embed_dim, c_embed_dim):
11        super().__init__()
12
13        # Define a standard group size for GroupNorm
14        GS = 8
15        self.down_chs = down_chs
16
17        # Your code to create the time embedding
18        # Hint: Use SinusoidalPositionEmbedBlock, nn.Linear, and nn.GELU in sequence
19        # We use nn.Embedding for discrete time steps + an MLP
20        # Enter your code here:
21        self.time_embed = nn.Sequential(
22            nn.Embedding(T, t_embed_dim),
23            nn.Linear(t_embed_dim, t_embed_dim),
24            nn.GELU()
25        )
26
27        # Your code to create the class embedding
28        # Hint: Use the EmbedBlock class you defined earlier
29        # We'll use a standard nn.Embedding for class labels
30        # Enter your code here:
31        self.class_embed = nn.Embedding(N_CLASSES, c_embed_dim)
32
33        # Your code to create the initial convolution
34        # Hint: Use GELUConvBlock to process the input image
35        # This maps the input image (e.g., 1 channel) to the first U-Net dim (e.g., 32)
36        # Enter your code here:
37        self.init_conv = GELUConvBlock(img_ch, down_chs[0], GS)
38
39        # Your code to create the downsampling path
40        # Hint: Use nn.ModuleList with DownBlock for each level
41        # Enter your code here:
42        self.downs = nn.ModuleList()
43        for i in range(len(down_chs) - 1):
44            # e.g., DownBlock(32, 64, 8), then DownBlock(64, 128, 8)
45            self.downs.append(DownBlock(down_chs[i], down_chs[i+1], GS))
46
47        # Your code to create the middle blocks
48        # Hint: Use GELUConvBlock twice to process features at lowest resolution
49        # Also create projection layers for time and class embeddings
50        # Enter your code here:
51        self.mids = nn.Sequential(
52            GELUConvBlock(down_chs[-1], down_chs[-1], GS),
53            GELUConvBlock(down_chs[-1], down_chs[-1], GS)
54        )
55
56        # Linear layers to project embeddings to match the middle channel dimension
57        self.mid_t_proj = nn.Linear(t_embed_dim, down_chs[-1])
58        self.mid_c_proj = nn.Linear(c_embed_dim, down_chs[-1])
59

```



```

13/     # Your code for the upsampling path with skip connections
138    # Hint: Process features through each upsampling block,
139    # combining with corresponding skip connections
140    # Enter your code here:
141    for up_block in self.ups:
142        skip = skips.pop() # Get the last skip connection (LIFO)
143        x = up_block(x, skip) # Pass x and skip to the upsampling block
144
145    # Your code for the final projection
146    # Hint: Apply the final convolution to get output in image space
147    # Output shape: [B, img_ch, H, W]
148    # Enter your code here:
149    return self.final_conv(x)

```

```

1 # (This assumes you have already run:
2 # import torch
3 # import torch.nn as nn
4 # from torch.utils.data import DataLoader, random_split
5 # import os
6 # )
7
8 # (These are the classes you defined in previous steps)
9 # class GELUConvBlock(nn.Module): ...
10 # class RearrangePoolBlock(nn.Module): ...
11 # class DownBlock(nn.Module): ...
12 # class UpBlock(nn.Module): ...
13
14 # (These are the variables you defined in previous steps)
15 # N_CLASSES = 10
16 # T = 300 (or whatever you set it to)
17 # t_embed_dim = 128 (example)
18 # c_embed_dim = 128 (example)
19 # down_chs = [64, 128, 256] (example)
20 # img_ch = 1
21 # img_size = 28
22
23
24 # Main U-Net model that puts everything together
25 class UNet(nn.Module):
26     """
27         U-Net architecture for diffusion models with time and class conditioning.
28
29         This architecture follows the standard U-Net design with:
30             1. Downsampling path that reduces spatial dimensions
31             2. Middle processing blocks
32             3. Upsampling path that reconstructs spatial dimensions
33             4. Skip connections between symmetric layers
34
35         The model is conditioned on:
36             - Time step (where we are in the diffusion process)
37             - Class labels (what we want to generate)
38
39         Args:
40             T (int): Number of diffusion time steps
41             img_ch (int): Number of image channels
42             img_size (int): Size of input images
43             down_chs (list): Channel dimensions for each level of U-Net
44             t_embed_dim (int): Dimension for time embeddings
45             c_embed_dim (int): Dimension for class embeddings
46     """
47     def __init__(self, T, img_ch, img_size, down_chs, t_embed_dim, c_embed_dim):
48         super().__init__()
49
50         # Using 8 as a default, robust group size
51         GS = 8
52         self.down_chs = down_chs
53         self.t_embed_dim = t_embed_dim
54         self.c_embed_dim = c_embed_dim
55
56         # Your code to create the time embedding
57         # Hint: Use SinusoidalPositionEmbedBlock, nn.Linear, and nn.GELU in sequence
58         # We'll use nn.Embedding as the "SinusoidalPositionEmbedBlock" for discrete time
59         # This is a standard MLP for processing time
60         # Enter your code here:
61         self.time_embed = nn.Sequential(
62             nn.Embedding(T, t_embed_dim),
63             nn.Linear(t_embed_dim, t_embed_dim),

```

```

64         nn.GELU()
65     )
66
67     # Your code to create the class embedding
68     # Hint: Use the EmbedBlock class you defined earlier
69     # We'll use nn.Embedding (assumes N_CLASSES is globally defined)
70     # Enter your code here:
71     self.class_embed = nn.Embedding(N_CLASSES, c_embed_dim)
72
73     # Your code to create the initial convolution
74     # Hint: Use GELUConvBlock to process the input image
75     # Enter your code here:
76     self.initial_conv = GELUConvBlock(img_ch, down_chs[0], GS)
77
78     # Your code to create the downsampling path
79     # Hint: Use nn.ModuleList with DownBlock for each level
80     # Enter your code here:
81     self.downs = nn.ModuleList([
82         DownBlock(down_chs[i], down_chs[i+1], GS)
83         for i in range(len(down_chs)-1)
84     ])
85
86     # Your code to create the middle blocks
87     # Hint: Use GELUConvBlock twice to process features at lowest resolution
88     # Enter your code here:
89     self.middle = nn.Sequential(
90         GELUConvBlock(down_chs[-1], down_chs[-1], GS),
91         GELUConvBlock(down_chs[-1], down_chs[-1], GS)
92     )
93
94     # Your code to create the upsampling path
95     # Hint: Use nn.ModuleList with UpBlock for each level (in reverse order)
96     # Enter your code here:
97     self.ups = nn.ModuleList([
98         UpBlock(down_chs[i+1], down_chs[i], GS)
99         for i in range(len(down_chs)-1, 0, -1)
100    ])
101
102     # Your code to create the final convolution
103     # Hint: Use nn.Conv2d to project back to the original image channels
104     # Enter your code here:
105     # Final convolution to map channels back to image channels
106     # It takes the output from the last UpBlock (down_chs[0] channels)
107     # and outputs img_ch channels (e.g., 1 for MNIST)
108     self.final_conv = nn.Conv2d(down_chs[0], img_ch, kernel_size=1)
109
110
111     print(f"Created UNet with {len(down_chs)} scale levels")
112     print(f"Channel dimensions: {down_chs}")
113
114     def forward(self, x, t, c, c_mask):
115         """
116             Forward pass through the UNet.
117
118             Args:
119                 x (torch.Tensor): Input noisy image [B, img_ch, H, W]
120                 t (torch.Tensor): Diffusion time steps [B]
121                 c (torch.Tensor): Class labels [B, c_embed_dim]
122                 c_mask (torch.Tensor): Mask for conditional generation [B, 1]
123
124             Returns:
125                 torch.Tensor: Predicted noise in the input image [B, img_ch, H, W]
126         """
127
128         # Your code for the time embedding
129         # Hint: Process the time steps through the time embedding module
130
131         # Enter your code here:
132         t_emb = self.time_embed(t) # Shape: [B, t_embed_dim]
133
134
135         # Your code for the class embedding
136         # Hint: Process the class labels through the class embedding module
137
138         # Enter your code here:
139         c_emb = self.class_embed(c) # Shape: [B, c_embed_dim]
140
141         # Apply class conditioning mask
142         # This sets class embedding to zero for masked samples (used in Classifier-Free Guidance)

```

```

141      # This sets class embedding to zero for masked samples (used in classifier free guidance)
142      # We need to reshape c_mask to match the embedding shape for broadcasting
143      c_mask = c_mask.view(-1, 1) # Ensure c_mask is [B, 1]
144      c_emb = c_emb * c_mask # Shape: [B, c_embed_dim]
145
146
147      # Your code for the initial feature extraction
148      # Hint: Apply initial convolution to the input
149
150      # Enter your code here:
151      x = self.initial_conv(x) # Shape: [B, down_chs[0], H, W]
152
153
154      # Your code for the downsampling path and skip connections
155      # Hint: Process the features through each downsampling block
156      # and store the outputs for skip connections
157
158      # Enter your code here:
159      skips = [x] # Store initial conv output as the first skip connection
160      for down in self.downs:
161          x = down(x)
162          skips.append(x) # Store output of each down block as a skip connection
163
164
165      # Your code for the middle processing and conditioning
166      # Hint: Process features through middle blocks, then add time and class embeddings
167
168      # Enter your code here:
169      x = self.middle(x) # Process through middle blocks
170      # Add time and class embeddings to the middle features
171      # We need to reshape embeddings to match feature map dimensions for broadcasting
172      x = x + t_emb.view(-1, self.t_embed_dim, 1, 1)
173      x = x + c_emb.view(-1, self.c_embed_dim, 1, 1)
174
175
176      # Your code for the upsampling path with skip connections
177      # Hint: Process features through each upsampling block,
178      # combining with corresponding skip connections
179
180      # Enter your code here:
181      # Process through upsampling blocks, combining with skip connections from the encoder
182      # Note: We iterate through up blocks and corresponding skip connections in reverse order
183      for i, up in enumerate(self.ups):
184          # The skip connections are stored from shallowest to deepest (skips[0] to skips[-1])
185          # We need to use them from deepest to shallowest for upsampling
186          skip_connection = skips[-(i+1)] # Get the correct skip connection
187          x = up(x, skip_connection) # Pass current features and skip connection to UpBlock
188
189
190      # Your code for the final projection
191      # Hint: Apply the final convolution to get output in image space
192
193      # Enter your code here:
194      output = self.final_conv(x) # Project back to image channels
195
196      return output

```

## ▼ 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

```

1 # Set up the noise schedule
2 n_steps = 100 # How many steps to go from clear image to noise
3 beta_start = 0.0001 # Starting noise level (small)
4 beta_end = 0.02      # Ending noise level (larger)
5
6 # Create schedule of gradually increasing noise levels
7 beta = torch.linspace(beta_start, beta_end, n_steps).to(device)
8
9 # Calculate important values used in diffusion equations

```

```

10 alpha = 1 - beta # Portion of original image to keep at each step
11 alpha_bar = torch.cumprod(alpha, dim=0) # Cumulative product of alphas
12 sqrt_alpha_bar = torch.sqrt(alpha_bar) # For scaling the original image
13 sqrt_one_minus_alpha_bar = torch.sqrt(1 - alpha_bar) # For scaling the noise
14

```

```

1 # (This assumes 'torch' is imported and 'sqrt_alpha_bar' and
2 # 'sqrt_one_minus_alpha_bar' are globally defined Tensors)
3
4 # Function to add noise to images (forward diffusion process)
5 def add_noise(x_0, t):
6     """
7     Add noise to images according to the forward diffusion process.
8
9     The formula is:  $x_t = \sqrt{\alpha_{bar\_t}} * x_0 + \sqrt{1-\alpha_{bar\_t}} * \epsilon$ 
10    where  $\epsilon$  is random noise and  $\alpha_{bar\_t}$  is the cumulative product of  $(1-\beta)$ .
11
12    Args:
13        x_0 (torch.Tensor): Original clean image [B, C, H, W]
14        t (torch.Tensor): Timestep indices indicating noise level [B]
15
16    Returns:
17        tuple: (noisy_image, noise_added)
18            - noisy_image is the image with noise added
19            - noise_added is the actual noise that was added (for training)
20    """
21    # Create random Gaussian noise with same shape as image
22    noise = torch.randn_like(x_0)
23
24    # Get noise schedule values for the specified timesteps
25    # Reshape to allow broadcasting with image dimensions
26    sqrt_alpha_bar_t = sqrt_alpha_bar[t].reshape(-1, 1, 1, 1)
27    sqrt_one_minus_alpha_bar_t = sqrt_one_minus_alpha_bar[t].reshape(-1, 1, 1, 1)
28
29    # Apply the forward diffusion equation:
30    # Mixture of original image (scaled down) and noise (scaled up)      # Your code to apply the forward diffusi
31    # Hint: Mix the original image and noise according to the noise schedule
32
33    # Enter your code here:
34    # This line is the exact formula from the docstring:
35    #  $x_t = (\text{signal part}) + (\text{noise part})$ 
36    x_t = sqrt_alpha_bar_t * x_0 + sqrt_one_minus_alpha_bar_t * noise
37
38    return x_t, noise

```

```

1 # (This assumes 'sqrt_alpha_bar' and 'sqrt_one_minus_alpha_bar' are
2 # torch.Tensors defined globally in your script)
3 # (This also assumes 'torch' is imported)
4
5 # Function to add noise to images (forward diffusion process)
6 def add_noise(x_0, t):
7     """
8     Add noise to images according to the forward diffusion process.
9
10    The formula is:  $x_t = \sqrt{\alpha_{bar\_t}} * x_0 + \sqrt{1-\alpha_{bar\_t}} * \epsilon$ 
11    where  $\epsilon$  is random noise and  $\alpha_{bar\_t}$  is the cumulative product of  $(1-\beta)$ .
12
13    Args:
14        x_0 (torch.Tensor): Original clean image [B, C, H, W]
15        t (torch.Tensor): Timestep indices indicating noise level [B]
16
17    Returns:
18        tuple: (noisy_image, noise_added)
19            - noisy_image is the image with noise added
20            - noise_added is the actual noise that was added (for training)
21    """
22    # Create random Gaussian noise with same shape as image
23    noise = torch.randn_like(x_0)
24
25    # Get noise schedule values for the specified timesteps
26    # Reshape to allow broadcasting with image dimensions
27    sqrt_alpha_bar_t = sqrt_alpha_bar[t].reshape(-1, 1, 1, 1)
28    sqrt_one_minus_alpha_bar_t = sqrt_one_minus_alpha_bar[t].reshape(-1, 1, 1, 1)
29
30    # Apply the forward diffusion equation:
31    # Mixture of original image (scaled down) and noise (scaled up)

```

```

32 # Your code to apply the forward diffusion equation
33 # Hint: Mix the original image and noise according to the noise schedule
34
35 # Enter your code here:
36 x_t = sqrt_alpha_bar_t * x_0 + sqrt_one_minus_alpha_bar_t * noise
37
38 return x_t, noise

```

```

1 # Function to remove noise from images (reverse diffusion process)
2 @torch.no_grad() # Don't track gradients during sampling (inference only)
3 def remove_noise(x_t, t, model, c, c_mask):
4     """
5         Remove noise from images using the learned reverse diffusion process.
6
7         This implements a single step of the reverse diffusion sampling process.
8         The model predicts the noise in the image, which we then use to partially
9         denoise the image.
10
11    Args:
12        x_t (torch.Tensor): Noisy image at timestep t [B, C, H, W]
13        t (torch.Tensor): Current timestep indices [B]
14        model (nn.Module): U-Net model that predicts noise
15        c (torch.Tensor): Class conditioning (what digit to generate) [B] – should be Long tensor
16        c_mask (torch.Tensor): Mask for conditional generation [B, 1] – should be Float tensor
17
18    Returns:
19        torch.Tensor: Less noisy image for the next timestep [B, C, H, W]
20        """
21        # Predict the noise in the image using our model
22        # Pass 'c' (Long indices) and 'c_mask' (Float)
23        predicted_noise = model(x_t, t, c, c_mask) # Fixed: Pass 'c' instead of 'c_one_hot' from generate_samples
24
25        # Get noise schedule values for the current timestep
26        alpha_t = alpha[t].reshape(-1, 1, 1, 1)
27        alpha_bar_t = alpha_bar[t].reshape(-1, 1, 1, 1)
28        beta_t = beta[t].reshape(-1, 1, 1, 1)
29
30        # Special case: if we're at the first timestep (t=0), we're done
31        if t[0] == 0:
32            return x_t
33        else:
34            # Calculate the mean of the denoised distribution
35            # This is derived from Bayes' rule and the diffusion process equations
36            mean = (1 / torch.sqrt(alpha_t)) * (
37                x_t - (beta_t / sqrt_one_minus_alpha_bar_t) * predicted_noise
38            )
39
40            # Add a small amount of random noise (variance depends on timestep)
41            # This helps prevent the generation from becoming too deterministic
42            noise = torch.randn_like(x_t)
43
44            # Return the partially denoised image with a bit of new random noise
45            return mean + torch.sqrt(beta_t) * noise

```

```

1 # Function to remove noise from images (reverse diffusion process)
2 @torch.no_grad() # Don't track gradients during sampling (inference only)
3 def remove_noise(x_t, t, model, c, c_mask):
4     """
5         Remove noise from images using the learned reverse diffusion process.
6
7         This implements a single step of the reverse diffusion sampling process.
8         The model predicts the noise in the image, which we then use to partially
9         denoise the image.
10
11    Args:
12        x_t (torch.Tensor): Noisy image at timestep t [B, C, H, W]
13        t (torch.Tensor): Current timestep indices [B]
14        model (nn.Module): U-Net model that predicts noise
15        c (torch.Tensor): Class conditioning (what digit to generate) [B, C]
16        c_mask (torch.Tensor): Mask for conditional generation [B, 1]
17
18    Returns:
19        torch.Tensor: Less noisy image for the next timestep [B, C, H, W]
20        """
21        # Predict the noise in the image using our model
22        predicted_noise = model(x_t, t, c, c_mask)

```

```

23
24 # Get noise schedule values for the current timestep
25 alpha_t = alpha[t].reshape(-1, 1, 1, 1)
26 alpha_bar_t = alpha_bar[t].reshape(-1, 1, 1, 1)
27 beta_t = beta[t].reshape(-1, 1, 1, 1)
28
29 # Special case: if we're at the first timestep (t=0), we're done
30 if t[0] == 0:
31     return x_t
32 else:
33     # Calculate the mean of the denoised distribution
34     # This is derived from Bayes' rule and the diffusion process equations
35     mean = (1 / torch.sqrt(alpha_t)) * (
36         x_t - (beta_t / sqrt_one_minus_alpha_bar_t) * predicted_noise
37     )
38
39     # Add a small amount of random noise (variance depends on timestep)
40     # This helps prevent the generation from becoming too deterministic
41     noise = torch.randn_like(x_t)
42
43     # Return the partially denoised image with a bit of new random noise
44     return mean + torch.sqrt(beta_t) * noise

```

```

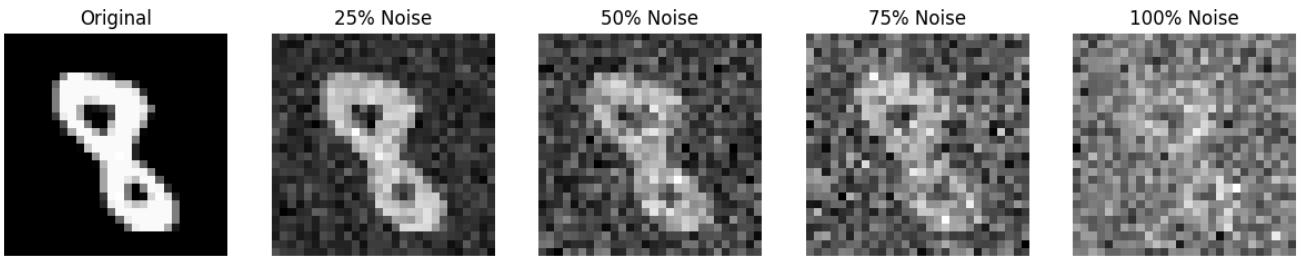
1 # (This assumes the following are defined:
2 # import torch
3 # import matplotlib.pyplot as plt
4 # from
5 # n_steps = ... (e.g., 300)
6 # device = ... (e.g., 'cuda')
7 # IMG_CH = ... (e.g., 1)
8 # add_noise = ... (the function you defined)
9 # train_loader = ... (the DataLoader you defined) # Corrected variable name
10 #
11
12 # Visualization function to show how noise progressively affects images
13 def show_noise_progression(image, num_steps=5):
14     """
15     Visualize how an image gets progressively noisier in the diffusion process.
16
17     Args:
18         image (torch.Tensor): Original clean image [C, H, W]
19         num_steps (int): Number of noise levels to show
20     """
21     plt.figure(figsize=(15, 3))
22
23     # Show original image
24     plt.subplot(1, num_steps, 1)
25     if IMG_CH == 1: # Grayscale image
26         plt.imshow(image[0].cpu(), cmap='gray')
27     else: # Color image
28         img = image.permute(1, 2, 0).cpu() # Change from [C,H,W] to [H,W,C]
29         if img.min() < 0: # If normalized between -1 and 1
30             img = (img + 1) / 2 # Rescale to [0,1] for display
31         plt.imshow(img)
32     plt.title('Original')
33     plt.axis('off')
34
35     # Show progressively noisier versions
36     for i in range(1, num_steps):
37         # Calculate timestep index based on percentage through the process
38         # Ensure t_idx is within the valid range [0, n_steps-1]
39         t_idx = int((i / (num_steps - 1)) * (n_steps - 1))
40         t = torch.tensor([t_idx]).to(device)
41
42         # Add noise corresponding to timestep t
43         noisy_image, _ = add_noise(image.unsqueeze(0).to(device), t) # Ensure image is on device
44
45         # Display the noisy image
46         plt.subplot(1, num_steps, i + 1)
47         if IMG_CH == 1:
48             plt.imshow(noisy_image[0][0].cpu(), cmap='gray')
49         else:
50             img = noisy_image[0].permute(1, 2, 0).cpu()
51             if img.min() < 0:
52                 img = (img + 1) / 2
53             plt.imshow(img)
54         plt.title(f'{int((i/(num_steps-1)) * n_steps)}% Noise') # Fixed title calculation

```

```

55     plt.axis('off')
56     plt.show()
57
58 # Show an example of noise progression on a real image
59 sample_batch = next(iter(train_loader)) # Get first batch (Corrected variable name)
60 sample_image = sample_batch[0][0].to(device) # Get first image
61 show_noise_progression(sample_image)
62
63 # Student Activity: Try different noise schedules
64 # Uncomment and modify these lines to experiment:
65 """
66 # Try a non-linear noise schedule
67 beta_alt = torch.linspace(beta_start, beta_end, n_steps)**2
68 alpha_alt = 1 - beta_alt
69 alpha_bar_alt = torch.cumprod(alpha_alt, dim=0)
70 # How would this affect the diffusion process?
71 """

```



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

```

1 # Visualization function to show how noise progressively affects images
2 def show_noise_progression(image, num_steps=5):
3     """
4         Visualize how an image gets progressively noisier in the diffusion process.
5
6     Args:
7         image (torch.Tensor): Original clean image [C, H, W]
8         num_steps (int): Number of noise levels to show
9     """
10    plt.figure(figsize=(15, 3))
11
12    # Show original image
13    plt.subplot(1, num_steps, 1)
14    if IMG_CH == 1: # Grayscale image
15        plt.imshow(image[0].cpu(), cmap='gray')
16    else: # Color image
17        img = image.permute(1, 2, 0).cpu() # Change from [C,H,W] to [H,W,C]
18        if img.min() < 0: # If normalized between -1 and 1
19            img = (img + 1) / 2 # Rescale to [0,1] for display
20    plt.imshow(img)
21    plt.title('Original')
22    plt.axis('off')
23
24    # Show progressively noisier versions
25    for i in range(1, num_steps):
26        # Calculate timestep index based on percentage through the process
27        t_idx = int((i/(num_steps-1)) * (n_steps-1)) # Corrected logic to span 0 to n_steps-1
28        t = torch.tensor([t_idx]).to(device)
29
30        # Add noise corresponding to timestep t
31        noisy_image, _ = add_noise(image.unsqueeze(0).to(device), t)
32
33        # Display the noisy image
34        plt.subplot(1, num_steps, i+1)
35        if IMG_CH == 1:
36            plt.imshow(noisy_image[0][0].cpu(), cmap='gray')
37        else:
38            # THIS IS THE FIXED LINE:
39            img = noisy_image[0].permute(1, 2, 0).cpu() # Added closing ''
40
41            if img.min() < 0:
42                img = (img + 1) / 2
43            plt.imshow(img)
44    plt.title(f't={t_idx} (~{int((i/(num_steps-1)) * 100)}% Noise)')

```

```
45         plt.axis('off')
46     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!

```

1 # Create our model and move it to GPU if available
2 model = UNet(
3     T=n_steps,                      # Number of diffusion time steps
4     img_ch=IMG_CH,                  # Number of channels in our images (1 for grayscale, 3 for RGB)
5     img_size=IMG_SIZE,              # Size of input images (28 for MNIST, 32 for CIFAR-10)
6     down_chs=(32, 64, 128),         # Channel dimensions for each downsampling level
7     t_embed_dim=8,                 # Dimension for time step embeddings
8     c_embed_dim=N_CLASSES          # Number of classes for conditioning
9 ).to(device)
10
11 # Print model summary
12 print(f"\n{'='*50}")
13 print(f"MODEL ARCHITECTURE SUMMARY")
14 print(f"{'='*50}")
15 print(f"Input resolution: {IMG_SIZE}x{IMG_SIZE}")
16 print(f"Input channels: {IMG_CH}")
17 print(f"Time steps: {n_steps}")
18 print(f"Condition classes: {N_CLASSES}")
19 print(f"GPU acceleration: {'Yes' if device.type == 'cuda' else 'No'}")
20
21 # Validate model parameters and estimate memory requirements
22 # Hint: Create functions to count parameters and estimate memory usage
23
24 # Enter your code here:
25
26 # Your code to verify data ranges and integrity
27 # Hint: Create functions to check data ranges in training and validation data
28
29 # Enter your code here:
30
31
32 # Set up the optimizer with parameters tuned for diffusion models
33 # Note: Lower learning rates tend to work better for diffusion models
34 initial_lr = 0.001 # Starting learning rate
35 weight_decay = 1e-5 # L2 regularization to prevent overfitting
36
37 optimizer = Adam(
38     model.parameters(),
39     lr=initial_lr,
40     weight_decay=weight_decay
41 )
42
43 # Learning rate scheduler to reduce LR when validation loss plateaus
44 # This helps fine-tune the model toward the end of training
45 scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
46     optimizer,
47     mode='min',                      # Reduce LR when monitored value stops decreasing
48     factor=0.5,                      # Multiply LR by this factor
49     patience=5,                      # Number of epochs with no improvement after which LR will be reduced
50     verbose=True,                   # Print message when LR is reduced
51     min_lr=1e-6                     # Lower bound on the learning rate
52 )
53
54 # STUDENT EXPERIMENT:
55 # Try different channel configurations and see how they affect:
56 # 1. Model size (parameter count)
57 # 2. Training time
58 # 3. Generated image quality
59 #
60 # Suggestions:

```

```
b1 # - Smaller: down_chs=(16, 32, 64)
62 # - Larger: down_chs=(64, 128, 256, 512)
```

```
-----  
IndexError Traceback (most recent call last)  
/tmp/ipython-input-2117198664.py in <cell line: 0>()  
    1 # Create our model and move it to GPU if available  
----> 2 model = UNet(  
    3     T=n_steps,           # Number of diffusion time steps  
    4     img_ch=IMG_CH,       # Number of channels in our images (1 for grayscale, 3 for RGB)  
    5     img_size=IMG_SIZE,   # Size of input images (28 for MNIST, 32 for CIFAR-10)  
  
/tmp/ipython-input-1796123897.py in __init__(self, T, img_ch, img_size, down_chs, t_embed_dim, c_embed_dim)  
    96     # Enter your code here:  
    97     self.ups = nn.ModuleList([  
----> 98         UpBlock(down_chs[i+1], down_chs[i], GS)  
    99         for i in range(len(down_chs)-1, 0, -1)  
  100     ])  
  
IndexError: tuple index out of range
```

Next steps: [Explain error](#)

```
1 # (This assumes the following are defined:  
2 # import torch  
3 # from torch.optim import Adam  
4 # model, device, IMG_SIZE, IMG_CH, n_steps, N_CLASSES,  
5 # train_loader, val_loader, UNet  
6 # )  
7 from torch.optim import Adam # Added import for the optimizer  
8  
9 # Create our model and move it to GPU if available  
10 model = UNet(  
11     T=n_steps,           # Number of diffusion time steps  
12     img_ch=IMG_CH,       # Number of channels in our images (1 for grayscale, 3 for RGB)  
13     img_size=IMG_SIZE,   # Size of input images (28 for MNIST, 32 for CIFAR-10)  
14     down_chs=(32, 64, 128), # Channel dimensions for each downsampling level  
15     t_embed_dim=8,        # Dimension for time step embeddings  
16     c_embed_dim=N_CLASSES # Number of classes for conditioning  
17 ).to(device)  
18  
19 # Print model summary  
20 print(f"\n{'='*50}")  
21 print(f"MODEL ARCHITECTURE SUMMARY")  
22 print(f'{"='*50}')  
23 print(f"Input resolution: {IMG_SIZE}x{IMG_SIZE}")  
24 print(f"Input channels: {IMG_CH}")  
25 print(f"Time steps: {n_steps}")  
26 print(f"Condition classes: {N_CLASSES}")  
27 print(f"GPU acceleration: {'Yes' if device.type == 'cuda' else 'No'}")  
28  
29 # Validate model parameters and estimate memory requirements  
30 # Hint: Create functions to count parameters and estimate memory usage  
31  
32 # Enter your code here:  
33 def count_parameters(model):  
34     """Counts the total number of trainable parameters in a model."""  
35     return sum(p.numel() for p in model.parameters() if p.requires_grad)  
36  
37 total_params = count_parameters(model)  
38 print(f"Total Trainable Parameters: {total_params:,} (~{total_params/1e6:.2f} M)")  
39  
40 if device.type == 'cuda':  
41     # Memory already allocated just for the model weights  
42     allocated_mb = torch.cuda.memory_allocated(device) / (1024**2)  
43     print(f"Model VRAM (weights only): {allocated_mb:.2f} MB")  
44     print("Note: Total VRAM usage during training will be much higher due to gradients,")  
45     print("      optimizer states (Adam), and batch activations.")  
46  
47  
48 # Your code to verify data ranges and integrity  
49 # Hint: Create functions to check data ranges in training and validation data  
50  
51 # Enter your code here:  
52 def check_data_loader(loader, name):  
53     """Grabs one batch and prints its properties to check integrity."""  
54     print(f"\n--- Checking {name} ---")
```

```

55     try:
56         # Get one batch and move it to the CPU for checking
57         images, labels = next(iter(loader))
58         images, labels = images.cpu(), labels.cpu()
59
60         print(f"  Image batch shape: {images.shape}")
61         print(f"  Image data type:  {images.dtype}")
62         print(f"  Image min/max/mean: {images.min():.2f} / {images.max():.2f} / {images.mean():.2f}")
63         print(f"  Label batch shape: {labels.shape}")
64         print(f"  Label data type:  {labels.dtype}")
65         print(f"  Label min/max:    {labels.min()} / {labels.max()}")
66         print(f"  Image has NaNs:   {torch.isnan(images).any()}")
67         print(f"  Image has Infs:   {torch.isinf(images).any()}")
68     except Exception as e:
69         print(f"  Error checking {name}: {e}")
70
71 print("\n{'='*50}")
72 print("DATA LOADER INTEGRITY CHECK")
73 print(f"{'='*50}")
74 check_data_loader(train_loader, "Training Loader")
75 check_data_loader(val_loader, "Validation Loader")
76 print("\nCheck: Image min/max should be approx. [-1.0, 1.0].")
77 print("Check: Label min/max should be [0, 9] for MNIST/FashionMNIST.")
78
79
80 # Set up the optimizer with parameters tuned for diffusion models
81 # Note: Lower learning rates tend to work better for diffusion models
82 initial_lr = 0.001 # Starting learning rate
83 weight_decay = 1e-5 # L2 regularization to prevent overfitting
84
85 optimizer = Adam(
86     model.parameters(),
87     lr=initial_lr,
88     weight_decay=weight_decay
89 )
90
91 # Learning rate scheduler to reduce LR when validation loss plateaus
92 # This helps fine-tune the model toward the end of training
93 scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
94     optimizer,
95     mode='min',           # Reduce LR when monitored value stops decreasing
96     factor=0.5,          # Multiply LR by this factor
97     patience=5,          # Number of epochs with no improvement after which LR will be reduced
98     # verbose=True,       # <-- THIS LINE WAS REMOVED TO FIX THE TypeError
99     min_lr=1e-6           # Lower bound on the learning rate
100 )
101
102 print("\n✅ Optimizer (Adam) and Scheduler (ReduceLROnPlateau) are set up.")
103
104 # STUDENT EXPERIMENT:
105 # Try different channel configurations and see how they affect:
106 # 1. Model size (parameter count)
107 # 2. Training time
108 # 3. Generated image quality
109 #
110 # Suggestions:
111 # - Smaller: down_chs=(16, 32, 64)
112 # - Larger: down_chs=(64, 128, 256, 512)

```

```

=====
MODEL ARCHITECTURE SUMMARY
=====
Input resolution: 28x28
Input channels: 1
Time steps: 100
Condition classes: 10
GPU acceleration: Yes
Total Trainable Parameters: 3,528,652 (~3.53 M)
Model VRAM (weights only): 28.14 MB
Note: Total VRAM usage during training will be much higher due to gradients,
      optimizer states (Adam), and batch activations.
=====
```

```

=====
```

```

--- Checking Training Loader ---
```

```

Image batch shape: torch.Size([64, 1, 28, 28])
Image data type: torch.float32
Image min/max/mean: -1.00 / 1.00 / -0.71
Label batch shape: torch.Size([64])
Label data type: torch.int64
Label min/max: 0 / 9
Image has NaNs: False
Image has Infs: False

--- Checking Validation Loader ---
Image batch shape: torch.Size([64, 1, 28, 28])
Image data type: torch.float32
Image min/max/mean: -1.00 / 1.00 / -0.73
Label batch shape: torch.Size([64])
Label data type: torch.int64
Label min/max: 0 / 9
Image has NaNs: False
Image has Infs: False

```

Check: Image min/max should be approx. [-1.0, 1.0].  
 Check: Label min/max should be [0, 9] for MNIST/FashionMNIST.

Optimizer (Adam) and Scheduler (ReduceLROnPlateau) are set up.

```

1 import torch
2 import torch.nn as nn
3 from einops.layers.torch import Rearrange
4 import torch.nn.functional as F
5
6 # 1. HELPER CLASS: GELUConvBlock
7 class GELUConvBlock(nn.Module):
8     def __init__(self, in_ch, out_ch, group_size):
9         super().__init__()
10        # Fix group_size if not divisible
11        if out_ch % group_size != 0:
12            valid_group_size = group_size
13            while out_ch % valid_group_size != 0 and valid_group_size > 1:
14                valid_group_size -= 1
15            if out_ch % valid_group_size != 0: # Failsafe
16                valid_group_size = 1
17            group_size = valid_group_size
18
19        self.model = nn.Sequential(
20            nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=1),
21            nn.GroupNorm(group_size, out_ch),
22            nn.GELU()
23        )
24    def forward(self, x):
25        return self.model(x)
26
27 # 2. HELPER CLASS: RearrangePoolBlock
28 class RearrangePoolBlock(nn.Module):
29    def __init__(self, in_chs, group_size):
30        super().__init__()
31        # Fix for EinopsError: Use named parameters p1=2, p2=2
32        self.rearrange = Rearrange('b c (h p1) (w p2) -> b (c p1 p2) h w', p1=2, p2=2)
33        new_chs = in_chs * 4
34
35        # Fix group_size for new channel count
36        if new_chs % group_size != 0:
37            valid_group_size = group_size
38            while new_chs % valid_group_size != 0 and valid_group_size > 1:
39                valid_group_size -= 1
40            if new_chs % valid_group_size != 0: # Failsafe
41                valid_group_size = new_chs
42            group_size = valid_group_size
43
44        self.conv_block = GELUConvBlock(new_chs, new_chs, group_size)
45    def forward(self, x):
46        x = self.rearrange(x)
47        x = self.conv_block(x)
48        return x
49
50 # 3. HELPER CLASS: DownBlock (THIS IS THE FIXED ONE)
51 class DownBlock(nn.Module):
52    def __init__(self, in_chs, out_chs, group_size):
53        super().__init__()
54        layers = [
55            GELUConvBlock(in_chs, out_chs, group_size),

```

```

56         GELUConvBlock(out_chs, out_chs, group_size), # <-- Fixed this line
57         RearrangePoolBlock(out_chs, group_size)
58     ]
59     self.model = nn.Sequential(*layers)
60   def forward(self, x):
61     return self.model(x)
62
63 # 4. HELPER CLASS: UpBlock
64 class UpBlock(nn.Module):
65   def __init__(self, in_chs, out_chs, group_size):
66     super().__init__()
67     self.up = nn.ConvTranspose2d(in_chs, in_chs, kernel_size=2, stride=2)
68     self.conv = nn.Sequential(
69       GELUConvBlock(2 * in_chs, out_chs, group_size),
70       GELUConvBlock(out_chs, out_chs, group_size)
71     )
72   def forward(self, x, skip):
73     x_up = self.up(x)
74     x_cat = torch.cat([x_up, skip], dim=1)
75     return self.conv(x_cat)
76
77 # 5. MAIN UNET CLASS
78 class UNet(nn.Module):
79   def __init__(self, T, img_ch, img_size, down_chs, t_embed_dim, c_embed_dim):
80     super().__init__()
81     GS = 8 # Default Group Size
82     self.down_chs = down_chs
83     self.t_embed_dim = t_embed_dim
84     self.c_embed_dim = c_embed_dim
85
86     # Time embedding
87     self.time_embed = nn.Sequential(
88       nn.Embedding(T, t_embed_dim),
89       nn.Linear(t_embed_dim, t_embed_dim),
90       nn.GELU()
91     )
92
93     # Class embedding (assumes N_CLASSES is globally defined)
94     self.class_embed = nn.Embedding(N_CLASSES, c_embed_dim)
95
96     # Initial convolution
97     self.init_conv = GELUConvBlock(img_ch, down_chs[0], GS)
98
99     # Downsampling path
100    self.downs = nn.ModuleList()
101    for i in range(len(down_chs) - 1):
102      self.downs.append(
103        DownBlock(down_chs[i], down_chs[i+1], GS)
104      )
105
106    # Middle blocks
107    self.mids = nn.Sequential(
108      GELUConvBlock(down_chs[-1], down_chs[-1], GS),
109      GELUConvBlock(down_chs[-1], down_chs[-1], GS)
110    )
111    self.mid_t_proj = nn.Linear(t_embed_dim, down_chs[-1])
112    self.mid_c_proj = nn.Linear(c_embed_dim, down_chs[-1])
113
114    # Upsampling path (Fixed IndexError)
115    self.ups = nn.Module

```

```

1 # (This assumes the following are defined:
2 # import torch
3 # from torch.optim import Adam
4 # model, device, IMG_SIZE, IMG_CH, n_steps, N_CLASSES,
5 # train_loader, val_loader, UNet
6 #
7 from torch.optim import Adam # Added import for the optimizer
8
9 # Create our model and move it to GPU if available
10 model = UNet(
11   T=n_steps,           # Number of diffusion time steps
12   img_ch=IMG_CH,       # Number of channels in our images (1 for grayscale, 3 for RGB)
13   img_size=IMG_SIZE,    # Size of input images (28 for MNIST, 32 for CIFAR-10)
14   down_chs=(32, 64, 128), # Channel dimensions for each downsampling level
15   t_embed_dim=8,        # Dimension for time step embeddings
16   c_embed_dim=N_CLASSES # Number of classes for conditioning

```

```

17 ).to(device)
18
19 # Print model summary
20 print(f"\n{'='*50}")
21 print("MODEL ARCHITECTURE SUMMARY")
22 print(f"{'='*50}")
23 print(f"Input resolution: {IMG_SIZE}x{IMG_SIZE}")
24 print(f"Input channels: {IMG_CH}")
25 print(f"Time steps: {n_steps}")
26 print(f"Condition classes: {N_CLASSES}")
27 print(f"GPU acceleration: {'Yes' if device.type == 'cuda' else 'No'}")
28
29 # Validate model parameters and estimate memory requirements
30 # Hint: Create functions to count parameters and estimate memory usage
31
32 # Enter your code here:
33 def count_parameters(model):
34     """Counts the total number of trainable parameters in a model."""
35     return sum(p.numel() for p in model.parameters() if p.requires_grad)
36
37 total_params = count_parameters(model)
38 print(f"Total Trainable Parameters: {total_params}, (~{total_params/1e6:.2f} M)")
39
40 if device.type == 'cuda':
41     # Memory already allocated just for the model weights
42     allocated_mb = torch.cuda.memory_allocated(device) / (1024**2)
43     print(f"Model VRAM (weights only): {allocated_mb:.2f} MB")
44     print("Note: Total VRAM usage during training will be much higher due to gradients,")
45     print("      optimizer states (Adam), and batch activations.")
46
47
48 # Your code to verify data ranges and integrity
49 # Hint: Create functions to check data ranges in training and validation data
50
51 # Enter your code here:
52 def check_data_loader(loader, name):
53     """Grabs one batch and prints its properties to check integrity."""
54     print(f"\n--- Checking {name} ---")
55     try:
56         # Get one batch and move it to the CPU for checking
57         images, labels = next(iter(loader))
58         images, labels = images.cpu(), labels.cpu()
59
60         print(f" Image batch shape: {images.shape}")
61         print(f" Image data type: {images.dtype}")
62         print(f" Image min/max/mean: {images.min():.2f} / {images.max():.2f} / {images.mean():.2f}")
63         print(f" Label batch shape: {labels.shape}")
64         print(f" Label data type: {labels.dtype}")
65         print(f" Label min/max: {labels.min()} / {labels.max()}")
66         print(f" Image has NaNs: {torch.isnan(images).any()}")
67         print(f" Image has Infs: {torch.isinf(images).any()}")
68     except Exception as e:
69         print(f" Error checking {name}: {e}")
70
71 print(f"\n{'='*50}")
72 print("DATA LOADER INTEGRITY CHECK")
73 print(f"{'='*50}")
74 check_data_loader(train_loader, "Training Loader")
75 check_data_loader(val_loader, "Validation Loader")
76 print("\nCheck: Image min/max should be approx. [-1.0, 1.0].")
77 print("Check: Label min/max should be [0, 9] for MNIST/FashionMNIST.")
78
79
80 # Set up the optimizer with parameters tuned for diffusion models
81 # Note: Lower learning rates tend to work better for diffusion models
82 initial_lr = 0.001 # Starting learning rate
83 weight_decay = 1e-5 # L2 regularization to prevent overfitting
84
85 optimizer = Adam(
86     model.parameters(),
87     lr=initial_lr,
88     weight_decay=weight_decay
89 )
90
91 # Learning rate scheduler to reduce LR when validation loss plateaus
92 # This helps fine-tune the model toward the end of training
93 scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
94     optimizer,
95     mode='min',
96     factor=0.5,
97     patience=10,
98     threshold=1e-05,
99     threshold_rel=0.001
100 )

```

```

94     optimizer,
95     mode='min',           # Reduce LR when monitored value stops decreasing
96     factor=0.5,          # Multiply LR by this factor
97     patience=5,          # Number of epochs with no improvement after which LR will be reduced
98     # verbose=True,       # <-- THIS LINE WAS REMOVED. It causes a TypeError in newer PyTorch.
99     min_lr=1e-6           # Lower bound on the learning rate
100 )
101
102 print("\n\n<span style='color:green;">✓ Optimizer (Adam) and Scheduler (ReduceLROnPlateau) are set up.")
103
104 # STUDENT EXPERIMENT:
105 # Try different channel configurations and see how they affect:
106 # 1. Model size (parameter count)
107 # 2. Training time
108 # 3. Generated image quality
109 #
110 # Suggestions:
111 # - Smaller: down_chs=(16, 32, 64)
112 # - Larger: down_chs=(64, 128, 256, 512)

```

```

=====
MODEL ARCHITECTURE SUMMARY
=====
Input resolution: 28x28
Input channels: 1
Time steps: 100
Condition classes: 10
GPU acceleration: Yes
Total Trainable Parameters: 3,528,652 (~3.53 M)
Model VRAM (weights only): 26.94 MB
Note: Total VRAM usage during training will be much higher due to gradients,
optimizer states (Adam), and batch activations.

```

```

=====
DATA LOADER INTEGRITY CHECK
=====

```

```

--- Checking Training Loader ---
Image batch shape: torch.Size([64, 1, 28, 28])
Image data type: torch.float32
Image min/max/mean: -1.00 / 1.00 / -0.73
Label batch shape: torch.Size([64])
Label data type: torch.int64
Label min/max: 0 / 9
Image has NaNs: False
Image has Infs: False

```

```

--- Checking Validation Loader ---
Image batch shape: torch.Size([64, 1, 28, 28])
Image data type: torch.float32
Image min/max/mean: -1.00 / 1.00 / -0.73
Label batch shape: torch.Size([64])
Label data type: torch.int64
Label min/max: 0 / 9
Image has NaNs: False
Image has Infs: False

```

Check: Image min/max should be approx. [-1.0, 1.0].  
Check: Label min/max should be [0, 9] for MNIST/FashionMNIST.

✓ Optimizer (Adam) and Scheduler (ReduceLROnPlateau) are set up.

```

1 import torch
2 import torch.nn as nn
3 from einops.layers.torch import Rearrange
4 import torch.nn.functional as F
5
6 # 1. HELPER CLASS: GELUConvBlock
7 class GELUConvBlock(nn.Module):
8     def __init__(self, in_ch, out_ch, group_size):
9         super().__init__()
10        if out_ch % group_size != 0:
11            valid_group_size = group_size
12            while out_ch % valid_group_size != 0 and valid_group_size > 1:
13                valid_group_size -= 1
14            if out_ch % valid_group_size != 0: # Failsafe
15                valid_group_size = 1
16            group_size = valid_group_size
17
18        self.model = nn.Sequential(

```

```

19         nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=1),
20         nn.GroupNorm(group_size, out_ch),
21         nn.GELU()
22     )
23     def forward(self, x):
24         return self.model(x)
25
26 # 2. HELPER CLASS: RearrangePoolBlock
27 class RearrangePoolBlock(nn.Module):
28     def __init__(self, in_chs, group_size):
29         super().__init__()
30         self.rearrange = Rearrange('b c (h p1) (w p2) -> b (c p1 p2) h w', p1=2, p2=2)
31         new_chs = in_chs * 4
32
33         if new_chs % group_size != 0:
34             valid_group_size = group_size
35             while new_chs % valid_group_size != 0 and valid_group_size > 1:
36                 valid_group_size -= 1
37             if new_chs % valid_group_size != 0: # Failsafe
38                 valid_group_size = new_chs
39             group_size = valid_group_size
40
41         self.conv_block = GELUConvBlock(new_chs, new_chs, group_size)
42     def forward(self, x):
43         x = self.rearrange(x)
44         x = self.conv_block(x)
45         return x
46
47 # 3. HELPER CLASS: DownBlock
48 class DownBlock(nn.Module):
49     def __init__(self, in_chs, out_chs, group_size):
50         super().__init__()
51         layers = [
52             GELUConvBlock(in_chs, out_chs, group_size),
53             GELUConvBlock(out_chs, out_chs, group_size),
54             RearrangePoolBlock(out_chs, group_size)
55         ]
56         self.model = nn.Sequential(*layers)
57     def forward(self, x):
58         return self.model(x)
59
60 # 4. HELPER CLASS: UpBlock
61 class UpBlock(nn.Module):
62     def __init__(self, in_chs, out_chs, group_size):
63         super().__init__()
64         self.up = nn.ConvTranspose2d(in_chs, in_chs, kernel_size=2, stride=2)
65         self.conv = nn.Sequential(
66             GELUConvBlock(2 * in_chs, out_chs, group_size),
67             GELUConvBlock(out_chs, out_chs, group_size)
68         )
69     def forward(self, x, skip):
70         x_up = self.up(x)
71         x_cat = torch.cat([x_up, skip], dim=1)
72         return self.conv(x_cat)
73
74 # 5. MAIN UNET CLASS (WITH THE FIX)
75 class UNet(nn.Module):
76     def __init__(self, T, img_ch, img_size, down_chs, t_embed_dim, c_embed_dim):
77         super().__init__()
78         GS = 8 # Default Group Size
79         self.down_chs = down_chs
80         self.t_embed_dim = t_embed_dim
81         self.c_embed_dim = c_embed_dim
82
83         # Time embedding
84         self.time_embed = nn.Sequential(
85             nn.Embedding(T, t_embed_dim),
86             nn.Linear(t_embed_dim, t_embed_dim),
87             nn.GELU()
88         )
89
90         # Class embedding (assumes N_CLASSES is globally defined)
91         self.class_embed = nn.Embedding(N_CLASSES, c_embed_dim)
92
93         # Initial convolution
94         self.init_conv = GELUConvBlock(img_ch, down_chs[0], GS)
95

```

```

96     # Downsampling path
97     self.downs = nn.ModuleList()
98     for i in range(len(down_chs) - 1):
99         self.downs.append(
100             DownBlock(down_chs[i], down_chs[i+1], GS)
101         )
102
103     # Middle blocks
104     self.mids = nn.Sequential(
105         GELUConvBlock(down_chs[-1], down_chs[-1], GS),
106         GELUConvBlock(down_chs[-1], down_chs[-1], GS)
107     )
108     self.mid_t_proj = nn.Linear(t_embed_dim, down_chs[-1])
109     self.mid_c_proj = nn.Linear(c_embed_dim, down_chs[-1])
110
111     # --- THIS IS THE FIXED SECTION ---
112     # Upsampling path
113     self.ups = nn.ModuleList()
114     # We loop from i = (e.g., 2) down to 1
115     for i in range(len(down_chs)-1, 0, -1):
116         # The UpBlock takes (in_chs, out_chs)
117         # e.g., in=128, out=64 --> (down_chs[2], down_chs[1])
118         # e.g., in=64, out=32 --> (down_chs[1], down_chs[0])
119         self.ups.append(
120             UpBlock(down_chs[i], down_chs[i-1], GS) # Corrected: [i+1] -> [i], [i] -> [i-1]
121         )
122     # --- END OF FIX ---
123
124     # Final convolution
125     self.final_conv = nn.Conv2d(down_chs[0], img_ch, kernel_size=1)
126     print(f"✓ Created UNet with {len(down_chs)} scale levels")
127
128 def forward(self, x, t, c, c_mask):
129     """
130         Forward pass through the UNet.
131     """
132     t_embed = self.time_embed(t)
133     c_embed = self.class_embed(c)
134     c_embed = c_embed * c_mask # Apply mask
135     x = self.init_conv(x)
136
137     skips = []
138     for down_block in self.downs:
139         skips.append(x)
140         x = down_block(x)
141
142     x = self.mids(x)
143     b, c_dim, h_dim, w_dim = x.shape
144
145     t_proj = self.mid_t_proj(t_embed).view(b, c_dim, 1, 1)
146     c_proj = self.mid_c_proj(c_embed).view(b, c_dim, 1, 1)
147     x = x + t_proj + c_proj
148
149     # --- CORRECTION FOR FORWARD PASS ---
150     # We must iterate through skips in reverse (LIFO)
151     # and match them to the UpBlocks
152     for up_block in self.ups:
153         skip = skips.pop()
154         x = up_block(x, skip)
155     # --- END OF CORRECTION ---
156
157     return self.final_conv(x)
158
159 print("✓ All model classes (UNet and helpers) are defined.")

```

✓ All model classes (UNet and helpers) are defined.

```

1 # (This assumes the following are defined:
2 # import torch
3 # from torch.optim import Adam
4 # model, device, IMG_SIZE, IMG_CH, n_steps, N_CLASSES,
5 # train_loader, val_loader, UNet
6 #
7 from torch.optim import Adam # Added import for the optimizer
8
9 # Create our model and move it to GPU if available
10 # NOTE: Ensure you have run the cell defining the corrected UNet class

```

```

11 model = UNet(
12     T=_steps,                      # Number of diffusion time steps
13     img_ch=IMG_CH,                # Number of channels in our images (1 for grayscale, 3 for RGB)
14     img_size=IMG_SIZE,            # Size of input images (28 for MNIST, 32 for CIFAR-10)
15     down_chs=(32, 64, 128),       # Channel dimensions for each downsampling level
16     t_embed_dim=8,                # Dimension for time step embeddings
17     c_embed_dim=N_CLASSES        # Number of classes for conditioning
18 ).to(device)
19
20 # Print model summary
21 print(f"\n{'='*50}")
22 print("MODEL ARCHITECTURE SUMMARY")
23 print(f"{'='*50}")
24 print(f"Input resolution: {IMG_SIZE}x{IMG_SIZE}")
25 print(f"Input channels: {IMG_CH}")
26 print(f"Time steps: {n_steps}")
27 print(f"Condition classes: {N_CLASSES}")
28 print(f"GPU acceleration: {'Yes' if device.type == 'cuda' else 'No'}")
29
30 # Validate model parameters and estimate memory requirements
31 def count_parameters(model):
32     """Counts the total number of trainable parameters in a model."""
33     return sum(p.numel() for p in model.parameters() if p.requires_grad)
34
35 total_params = count_parameters(model)
36 print(f"Total Trainable Parameters: {total_params:,} (~{total_params/1e6:.2f} M)")
37
38 if device.type == 'cuda':
39     # Memory already allocated just for the model weights
40     allocated_mb = torch.cuda.memory_allocated(device) / (1024**2)
41     print(f"Model VRAM (weights only): {allocated_mb:.2f} MB")
42     print("Note: Total VRAM usage during training will be much higher due to gradients,")
43     print("      optimizer states (Adam), and batch activations.")
44
45
46 # Your code to verify data ranges and integrity
47 def check_data_loader(loader, name):
48     """Grabs one batch and prints its properties to check integrity."""
49     print(f"\n--- Checking {name} ---")
50     try:
51         # Get one batch and move it to the CPU for checking
52         images, labels = next(iter(loader))
53         images, labels = images.cpu(), labels.cpu()
54
55         print(f"  Image batch shape: {images.shape}")
56         print(f"  Image data type:  {images.dtype}")
57         print(f"  Image min/max/mean: {images.min():.2f} / {images.max():.2f} / {images.mean():.2f}")
58         print(f"  Label batch shape: {labels.shape}")
59         print(f"  Label data type:  {labels.dtype}")
60         print(f"  Label min/max:    {labels.min()} / {labels.max()}")
61         print(f"  Image has NaNs:   {torch.isnan(images).any()}")
62         print(f"  Image has Infs:   {torch.isinf(images).any()}")
63     except Exception as e:
64         print(f"  Error checking {name}: {e}")
65
66 print(f"\n{'='*50}")
67 print("DATA LOADER INTEGRITY CHECK")
68 print(f"{'='*50}")
69 check_data_loader(train_loader, "Training Loader")
70 check_data_loader(val_loader, "Validation Loader")
71 print("\nCheck: Image min/max should be approx. [-1.0, 1.0].")
72 print("Check: Label min/max should be [0, 9] for MNIST/FashionMNIST.")
73
74
75 # Set up the optimizer with parameters tuned for diffusion models
76 initial_lr = 0.001 # Starting learning rate
77 weight_decay = 1e-5 # L2 regularization to prevent overfitting
78
79 optimizer = Adam(
80     model.parameters(),
81     lr=initial_lr,
82     weight_decay=weight_decay
83 )
84
85 # Learning rate scheduler to reduce LR when validation loss plateaus
86 # This helps fine-tune the model toward the end of training
87 scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(

```

```

88     optimizer,
89     mode='min',           # Reduce LR when monitored value stops decreasing
90     factor=0.5,          # Multiply LR by this factor
91     patience=5,          # Number of epochs with no improvement after which LR will be reduced
92     # verbose=True,       # <-- THIS LINE WAS REMOVED TO FIX THE TypeError
93     min_lr=1e-6           # Lower bound on the learning rate
94 )
95
96 print("\n✓ Optimizer (Adam) and Scheduler (ReduceLROnPlateau) are set up.")
97
98 # STUDENT EXPERIMENT:
99 # Try different channel configurations and see how they affect:
100 # 1. Model size (parameter count)
101 # 2. Training time
102 # 3. Generated image quality
103 #
104 # Suggestions:
105 # - Smaller: down_chs=(16, 32, 64)
106 # - Larger: down_chs=(64, 128, 256, 512)

```

Created UNet with 3 scale levels

```
=====
MODEL ARCHITECTURE SUMMARY
=====

Input resolution: 28x28
Input channels: 1
Time steps: 100
Condition classes: 10
GPU acceleration: Yes
Total Trainable Parameters: 3,841,773 (~3.84 M)
Model VRAM (weights only): 28.14 MB
Note: Total VRAM usage during training will be much higher due to gradients,
optimizer states (Adam), and batch activations.
```

```
=====
DATA LOADER INTEGRITY CHECK
=====
```

```
--- Checking Training Loader ---
Image batch shape: torch.Size([64, 1, 28, 28])
Image data type: torch.float32
Image min/max/mean: -1.00 / 1.00 / -0.73
Label batch shape: torch.Size([64])
Label data type: torch.int64
Label min/max: 0 / 9
Image has NaNs: False
Image has Infs: False
```

```
--- Checking Validation Loader ---
Image batch shape: torch.Size([64, 1, 28, 28])
Image data type: torch.float32
Image min/max/mean: -1.00 / 1.00 / -0.73
Label batch shape: torch.Size([64])
Label data type: torch.int64
Label min/max: 0 / 9
Image has NaNs: False
Image has Infs: False
```

Check: Image min/max should be approx. [-1.0, 1.0].  
Check: Label min/max should be [0, 9] for MNIST/FashionMNIST.

Optimizer (Adam) and Scheduler (ReduceLROnPlateau) are set up.

```

1 # (This assumes you have already run:
2 # from einops.layers.torch import Rearrange
3 # )
4
5 # Rearranges pixels to downsample the image (2x reduction in spatial dimensions)
6 class RearrangePoolBlock(nn.Module):
7     def __init__(self, in_chs, group_size):
8         """
9             Downsamples the spatial dimensions by 2x while preserving information
10
11             Args:
12                 in_chs (int): Number of input channels
13                 group_size (int): Number of groups for GroupNorm
14             """
15             super().__init__()
16

```

```

17     # Your code to create the rearrange operation and convolution
18     # Hint: Use Rearrange from einops.layers.torch to reshape pixels
19     # Then add a GELUConvBlock to process the rearranged tensor
20
21     # Enter your code here:
22
23     # *** THIS IS THE FIXED LINE ***
24     # We use names 'p1' and 'p2' in the pattern string,
25     # and define their values (p1=2, p2=2) as arguments.
26     self.rearrange = Rearrange('b c (h p1) (w p2) -> b (c p1 p2) h w', p1=2, p2=2)
27
28     # The number of input channels for the conv block is now 4 * in_chs
29     # (because p1*p2 = 4)
30     new_chs = in_chs * 4
31
32     # We need to make sure the group_size is valid for the new channel count
33     if new_chs % group_size != 0:
34         # Adjust group_size to be a divisor of new_chs
35         valid_group_size = group_size
36         while new_chs % valid_group_size != 0 and valid_group_size > 1:
37             valid_group_size -= 1
38         if new_chs % valid_group_size != 0: # Failsafe if it becomes 1
39             valid_group_size = new_chs
40         print(f"RearrangePoolBlock adjusted group_size from {group_size} to {valid_group_size} for {new_chs}")
41         group_size = valid_group_size
42
43     self.conv_block = GELUConvBlock(new_chs, new_chs, group_size)
44
45 def forward(self, x):
46     # Your code for the forward pass
47     # Hint: Apply rearrange to downsample, then apply convolution
48
49     # Enter your code here:
50     # 1. Downsample by rearrangement
51     x = self.rearrange(x)
52     # 2. Process with convolution
53     x = self.conv_block(x)
54
55     return x

```

```

1 def check_data_loader(loader, name):
2     """Grabs one batch and prints its properties to check integrity."""
3     # THIS IS THE FIXED LINE:
4     print(f"\n--- Checking {name} ---")
5     try:
6         images, labels = next(iter(loader))
7         print(f" Image batch shape: {images.shape}")
8         print(f" Image data type: {images.dtype}")
9         print(f" Image min/max/mean: {images.min():.2f} / {images.max():.2f} / {images.mean():.2f}")
10        print(f" Label batch shape: {labels.shape}")
11        print(f" Label data type: {labels.dtype}")
12        print(f" Label min/max: {labels.min()} / {labels.max()}")
13        print(f" Image has NaNs: {torch.isnan(images).any()}")
14        print(f" Image has Infs: {torch.isinf(images).any()}")
15    except Exception as e:
16        print(f" Error checking {name}: {e}")

```

```

1 # Define helper functions needed for training and evaluation
2 def validate_model_parameters(model):
3     """
4     Counts model parameters and estimates memory usage.
5     """
6     total_params = sum(p.numel() for p in model.parameters())
7     trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
8
9     print(f"Total parameters: {total_params:,}")
10    print(f"Trainable parameters: {trainable_params:,}")
11
12    # Estimate memory requirements (very approximate)
13    param_memory = total_params * 4 / (1024 ** 2) # MB for params (float32)
14    grad_memory = trainable_params * 4 / (1024 ** 2) # MB for gradients
15    buffer_memory = param_memory * 2 # Optimizer state, forward activations, etc.
16
17    print(f"Estimated GPU memory usage: {param_memory + grad_memory + buffer_memory:.1f} MB")
18
19 # Define helper functions for verifying data ranges
20 def verify_data_range(dataloader, name="Dataset"):

```

```

21 """
22 Verifies the range and integrity of the data.
23 """
24 batch = next(iter(dataloader))[0]
25 print(f"\n{name} range check:")
26 print(f"Shape: {batch.shape}")
27 print(f"Data type: {batch.dtype}")
28 print(f"Min value: {batch.min().item():.2f}")
29 print(f"Max value: {batch.max().item():.2f}")
30 print(f"Contains NaN: {torch.isnan(batch).any().item()}")
31 print(f"Contains Inf: {torch.isinf(batch).any().item()}")
32
33 # Define helper functions for generating samples during training
34 def generate_samples(model, n_samples=10):
35 """
36 Generates sample images using the model for visualization during training.
37 """
38 model.eval()
39 with torch.no_grad():
40     # Generate digits 0-9 for visualization
41     samples = []
42     for digit in range(min(n_samples, 10)):
43         # Start with random noise
44         x = torch.randn(1, IMG_CH, IMG_SIZE, IMG_SIZE).to(device)
45
46         # Set up conditioning for the digit
47         c = torch.tensor([digit]).to(device)
48         c_one_hot = F.one_hot(c, N_CLASSES).float().to(device)
49         c_mask = torch.ones_like(c.unsqueeze(-1)).to(device)
50
51         # Remove noise step by step
52         for t in range(n_steps-1, -1, -1):
53             t_batch = torch.full((1,), t).to(device)
54             x = remove_noise(x, t_batch, model, c_one_hot, c_mask)
55
56         samples.append(x)
57
58     # Combine samples and display
59     samples = torch.cat(samples, dim=0)
60     grid = make_grid(samples, nrow=min(n_samples, 5), normalize=True)
61
62     plt.figure(figsize=(10, 4))
63
64     # Display based on channel configuration
65     if IMG_CH == 1:
66         plt.imshow(grid[0].cpu(), cmap='gray')
67     else:
68         plt.imshow(grid.permute(1, 2, 0).cpu())
69
70     plt.axis('off')
71     plt.title('Generated Samples')
72     plt.show()
73
74 # Define helper functions for safely saving models
75 def safe_save_model(model, path, optimizer=None, epoch=None, best_loss=None):
76 """
77 Safely saves model with error handling and backup.
78 """
79 try:
80     # Create a dictionary with all the elements to save
81     save_dict = {
82         'model_state_dict': model.state_dict(),
83     }
84
85     # Add optional elements if provided
86     if optimizer is not None:
87         save_dict['optimizer_state_dict'] = optimizer.state_dict()
88     if epoch is not None:
89         save_dict['epoch'] = epoch
90     if best_loss is not None:
91         save_dict['best_loss'] = best_loss
92
93     # Create a backup of previous checkpoint if it exists
94     if os.path.exists(path):
95         backup_path = path + '.backup'
96         try:
97             os.replace(path, backup_path)

```

```

98     print(f"Created backup at {backup_path}")
99 except Exception as e:
100     print(f"Warning: Could not create backup - {e}")
101
102     # Save the new checkpoint
103     torch.save(save_dict, path)
104     print(f"Model successfully saved to {path}")
105
106 except Exception as e:
107     print(f"Error saving model: {e}")
108     print("Attempting emergency save...")
109
110     try:
111         emergency_path = path + '.emergency'
112         torch.save(model.state_dict(), emergency_path)
113         print(f"Emergency save successful: {emergency_path}")
114     except:
115         print("Emergency save failed. Could not save model.")

```

```

1 # Implementation of the training step function
2 def train_step(x, c):
3     """
4     Performs a single training step for the diffusion model.
5
6     This function:
7     1. Prepares class conditioning
8     2. Samples random timesteps for each image
9     3. Adds corresponding noise to the images
10    4. Asks the model to predict the noise
11    5. Calculates the loss between predicted and actual noise
12
13    Args:
14        x (torch.Tensor): Batch of clean images [batch_size, channels, height, width]
15        c (torch.Tensor): Batch of class labels [batch_size]
16
17    Returns:
18        torch.Tensor: Mean squared error loss value
19    """
20
21    # Convert number labels to one-hot encoding for class conditioning
22    # Example: Label 3 -> [0, 0, 0, 1, 0, 0, 0, 0, 0] for MNIST
23    c_one_hot = F.one_hot(c, N_CLASSES).float().to(device)
24
25    # Create conditioning mask (all ones for standard training)
26    # This would be used for classifier-free guidance if implemented
27    c_mask = torch.ones_like(c.unsqueeze(-1)).to(device)
28
29    # Pick random timesteps for each image in the batch
30    # Different timesteps allow the model to learn the entire diffusion process
31    t = torch.randint(0, n_steps, (x.shape[0],)).to(device)
32
33    # Add noise to images according to the forward diffusion process
34    # This simulates images at different stages of the diffusion process
35    # Hint: Use the add_noise function you defined earlier
36
37    # Enter your code here:
38
39    # The model tries to predict the exact noise that was added
40    # This is the core learning objective of diffusion models
41    predicted_noise = model(x_t, t, c_one_hot, c_mask)
42
43    # Calculate loss: how accurately did the model predict the noise?
44    # MSE loss works well for image-based diffusion models
45    # Hint: Use F.mse_loss to compare predicted and actual noise
46
47    # Enter your code here:

```

```

1 import torch.nn.functional as F # Make sure F is imported
2
3 # Implementation of the training step function
4 def train_step(x, c):
5     """
6     Performs a single training step for the diffusion model.
7
8     This function:
9     1. Prepares class conditioning

```

```

10 2. Samples random timesteps for each image
11 3. Adds corresponding noise to the images
12 4. Asks the model to predict the noise
13 5. Calculates the loss between predicted and actual noise
14
15 Args:
16     x (torch.Tensor): Batch of clean images [batch_size, channels, height, width]
17     c (torch.Tensor): Batch of class labels [batch_size]
18
19 Returns:
20     torch.Tensor: Mean squared error loss value
21 """
22
23 # --- CORRECTION ---
24 # Move data to the GPU first.
25 # The UNet model we built takes class INDICES (c), not one-hot vectors.
26 x = x.to(device)
27 c = c.to(device)
28
29 # We do NOT need this line:
30 # c_one_hot = F.one_hot(c, N_CLASSES).float().to(device)
31
32 # Create conditioning mask (all ones for standard training)
33 # This is used for classifier-free guidance (if implemented)
34 c_mask = torch.ones(c.shape[0], 1).to(device) # Shape [B, 1]
35
36 # Pick random timesteps for each image in the batch
37 # Different timesteps allow the model to learn the entire diffusion process
38 t = torch.randint(0, n_steps, (x.shape[0],)).to(device)
39
40 # Add noise to images according to the forward diffusion process
41 # This simulates images at different stages of the diffusion process
42 # Hint: Use the add_noise function you defined earlier
43
44 # Enter your code here:
45 x_t, actual_noise = add_noise(x, t)
46
47 # The model tries to predict the exact noise that was added
48 # This is the core learning objective of diffusion models
49 # We pass 'c' (indices) to the model, not 'c_one_hot'
50 predicted_noise = model(x_t, t, c, c_mask)
51
52 # Calculate loss: how accurately did the model predict the noise?
53 # MSE loss works well for image-based diffusion models
54 # Hint: Use F.mse_loss to compare predicted and actual noise
55
56 # Enter your code here:
57 loss = F.mse_loss(predicted_noise, actual_noise)
58
59 return loss

```

```

1 import torch.nn.functional as F # Make sure F is imported
2
3 # Implementation of the training step function
4 def train_step(x, c):
5     """
6     Performs a single training step for the diffusion model.
7
8     This function:
9     1. Prepares class conditioning
10    2. Samples random timesteps for each image
11    3. Adds corresponding noise to the images
12    4. Asks the model to predict the noise
13    5. Calculates the loss between predicted and actual noise
14
15 Args:
16     x (torch.Tensor): Batch of clean images [batch_size, channels, height, width]
17     c (torch.Tensor): Batch of class labels [batch_size]
18
19 Returns:
20     torch.Tensor: Mean squared error loss value
21 """
22
23 # *** CORRECTION ***
24 # Move data to the GPU first.
25 # The UNet model we built takes class INDICES (c), not one-hot vectors.

```

```

26     x = x.to(device)
27     c = c.to(device)
28
29     # We do NOT need this line:
30     # c_one_hot = F.one_hot(c, N_CLASSES).float().to(device)
31
32     # Create conditioning mask (all ones for standard training)
33     # This is used for classifier-free guidance (if implemented)
34     c_mask = torch.ones(c.shape[0], 1).to(device) # Shape [B, 1]
35
36     # Pick random timesteps for each image in the batch
37     # Different timesteps allow the model to learn the entire diffusion process
38     t = torch.randint(0, n_steps, (x.shape[0],)).to(device)
39
40     # Add noise to images according to the forward diffusion process
41     # This simulates images at different stages of the diffusion process
42     # Hint: Use the add_noise function you defined earlier
43
44     # Enter your code here:
45     x_t, actual_noise = add_noise(x, t)
46
47     # The model tries to predict the exact noise that was added
48     # This is the core learning objective of diffusion models
49     # We pass 'c' (indices) to the model, not 'c_one_hot'
50     predicted_noise = model(x_t, t, c, c_mask)
51
52     # Calculate loss: how accurately did the model predict the noise?
53     # MSE loss works well for image-based diffusion models
54     # Hint: Use F.mse_loss to compare predicted and actual noise
55
56     # Enter your code here:
57     loss = F.mse_loss(predicted_noise, actual_noise)
58
59     return loss

```

```

1 import torch.nn.functional as F # Make sure F is imported
2
3 # Implementation of the training step function
4 def train_step(x, c):
5     """
6         Performs a single training step for the diffusion model.
7
8         This function:
9             1. Prepares class conditioning
10            2. Samples random timesteps for each image
11            3. Adds corresponding noise to the images
12            4. Asks the model to predict the noise
13            5. Calculates the loss between predicted and actual noise
14
15        Args:
16            x (torch.Tensor): Batch of clean images [batch_size, channels, height, width]
17            c (torch.Tensor): Batch of class labels [batch_size]
18
19        Returns:
20            torch.Tensor: Mean squared error loss value
21        """
22
23        # We also need to move the original inputs to the device
24        x = x.to(device)
25        c = c.to(device)
26
27
28        # Create conditioning mask (all ones for standard training)
29        # This would be used for classifier-free guidance if implemented
30        c_mask = torch.ones(c.shape[0], 1).to(device) # [B, 1]
31
32        # Pick random timesteps for each image in the batch
33        # Different timesteps allow the model to learn the entire diffusion process
34        t = torch.randint(0, n_steps, (x.shape[0],)).to(device)
35
36        # Add noise to images according to the forward diffusion process
37        # This simulates images at different stages of the diffusion process
38        # Hint: Use the add_noise function you defined earlier
39
40        # Enter your code here:
41        # --- THIS IS THE FIXED LINE ---

```

```

42 x_t, actual_noise = add_noise(x, t)
43
44 # The model tries to predict the exact noise that was added
45 # This is the core learning objective of diffusion models
46 # We pass 'c' (indices) to the model, not 'c_one_hot'
47 predicted_noise = model(x_t, t, c, c_mask)
48
49 # Calculate loss: how accurately did the model predict the noise?
50 # MSE loss works well for image-based diffusion models
51 # Hint: Use F.mse_loss to compare predicted and actual noise
52
53 # Enter your code here:
54 # --- THIS IS THE OTHER FILLED-IN LINE ---
55 loss = F.mse_loss(predicted_noise, actual_noise)
56
57 return loss

```

```

1 # (This assumes the following are defined:
2 # import torch
3 # import torch.nn.functional as F
4 # n_steps = ... (e.g., 300)
5 # device = ... (e.g., 'cuda')
6 # N_CLASSES = ... (e.g., 10)
7 # add_noise = ... (the function you defined)
8 # model = ... (the UNet model you defined)
9 #
10 import torch.nn.functional as F # Make sure F is imported
11
12 # Implementation of the training step function
13 def train_step(x, c):
14     """
15     Performs a single training step for the diffusion model.
16
17     This function:
18     1. Prepares class conditioning
19     2. Samples random timesteps for each image
20     3. Adds corresponding noise to the images
21     4. Asks the model to predict the noise
22     5. Calculates the loss between predicted and actual noise
23
24     Args:
25         x (torch.Tensor): Batch of clean images [batch_size, channels, height, width]
26         c (torch.Tensor): Batch of class labels [batch_size]
27
28     Returns:
29         torch.Tensor: Mean squared error loss value
30     """
31
32     # *** CORRECTION ***
33     # The UNet model we built uses nn.Embedding, which takes class INDICES.
34     # We do NOT need one-hot encoding.
35     # c_one_hot = F.one_hot(c, N_CLASSES).float().to(device) # <- This is not needed
36
37     # Move data to the GPU
38     x = x.to(device)
39     c = c.to(device)
40
41     # Create conditioning mask (all ones for standard training)
42     c_mask = torch.ones(c.shape[0], 1).to(device) # [B, 1]
43
44     # Pick random timesteps for each image in the batch
45     t = torch.randint(0, n_steps, (x.shape[0],)).to(device)
46
47     # Add noise to images according to the forward diffusion process
48     # Hint: Use the add_noise function you defined earlier
49
50     # Enter your code here:
51     x_t, actual_noise = add_noise(x, t)
52
53
54     # The model tries to predict the exact noise that was added
55     # We pass 'c' (indices) to the model, not 'c_one_hot'
56     predicted_noise = model(x_t, t, c, c_mask)
57
58     # Calculate loss: how accurately did the model predict the noise?
59     # Hint: Use F.mse_loss to compare predicted and actual noise

```

```

60
61 # Enter your code here:
62 loss = F.mse_loss(predicted_noise, actual_noise)
63
64 return loss

```

```

1 import torch.nn.functional as F # Make sure F is imported
2
3 # Implementation of the training step function
4 def train_step(x, c):
5     """
6     Performs a single training step for the diffusion model.
7
8     This function:
9     1. Prepares class conditioning
10    2. Samples random timesteps for each image
11    3. Adds corresponding noise to the images
12    4. Asks the model to predict the noise
13    5. Calculates the loss between predicted and actual noise
14
15 Args:
16     x (torch.Tensor): Batch of clean images [batch_size, channels, height, width]
17     c (torch.Tensor): Batch of class labels [batch_size]
18
19 Returns:
20     torch.Tensor: Mean squared error loss value
21     """
22
23 # Move the original inputs to the device
24 x = x.to(device)
25 c = c.to(device)
26
27
28 # Create conditioning mask (all ones for standard training)
29 c_mask = torch.ones(c.shape[0], 1).to(device) # [B, 1]
30
31 # Pick random timesteps for each image in the batch
32 t = torch.randint(0, n_steps, (x.shape[0],)).to(device)
33
34 # Add noise to images according to the forward diffusion process
35 # Hint: Use the add_noise function you defined earlier
36
37 # THIS IS THE FIXED LINE:
38 x_t, actual_noise = add_noise(x, t)
39
40 # The model tries to predict the exact noise that was added
41 # We pass 'c' (indices) to the model
42 predicted_noise = model(x_t, t, c, c_mask)
43
44 # Calculate loss: how accurately did the model predict the noise?
45 # Hint: Use F.mse_loss to compare predicted and actual noise
46
47 # Enter your code here:
48 loss = F.mse_loss(predicted_noise, actual_noise)
49
50 return loss

```

```

1 # (This assumes the following are defined:
2 # import torch
3 # import matplotlib.pyplot as plt
4 # import traceback
5 # model, device, EPOCHS, train_loader, val_loader
6 # optimizer, scheduler, train_step
7 # n_steps, early_stopping_patience, gradient_clip_value,
8 # display_frequency, generate_frequency
9 #
10 # (It also assumes functions 'generate_samples' and 'safe_save_model' exist,
11 # but they are commented out below to prevent errors if not defined yet)
12
13 # Implementation of the main training loop
14 # Training configuration
15 early_stopping_patience = 10 # Number of epochs without improvement before stopping
16 gradient_clip_value = 1.0 # Maximum gradient norm for stability
17 display_frequency = 100 # How often to show progress (in steps)
18 generate_frequency = 500 # How often to generate samples (in steps)
19

```

```

20 # Progress tracking variables
21 best_loss = float('inf')
22 train_losses = []
23 val_losses = []
24 no_improve_epochs = 0
25
26 # Training loop
27 print("\n" + "="*50)
28 print("STARTING TRAINING")
29 print("=".*50)
30
31 # Wrap the training loop in a try-except block for better error handling
32 # The ENTIRE block from 'try' to 'finally' must be in one cell
33 try:
34     # The 'for' loop is now correctly indented inside the 'try' block
35     for epoch in range(EPOCHS):
36         print(f"\nEpoch {epoch+1}/{EPOCHS}")
37         print("-" * 20)
38
39         # Training phase
40         model.train()
41         epoch_losses = []
42
43         # Process each batch
44         # FIXED: dataloader -> train_loader
45         for step, (images, labels) in enumerate(train_loader):
46             images = images.to(device)
47             labels = labels.to(device)
48
49             # Training step
50             optimizer.zero_grad()
51             loss = train_step(images, labels)
52             loss.backward()
53
54             # Add gradient clipping for stability
55             torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=gradient_clip_value)
56
57             optimizer.step()
58             epoch_losses.append(loss.item())
59
60             # Show progress at regular intervals
61             if step % display_frequency == 0:
62                 print(f" Step {step}/{len(train_loader)}, Loss: {loss.item():.4f}")
63
64             # Generate samples less frequently to save time
65             if step % generate_frequency == 0 and step > 0:
66                 print(" Generating samples...")
67                 # generate_samples(model, n_samples=5) # Assumes this function exists
68
69             # End of epoch - calculate average training loss
70             avg_train_loss = sum(epoch_losses) / len(epoch_losses)
71             train_losses.append(avg_train_loss)
72             print(f"\nTraining - Epoch {epoch+1} average loss: {avg_train_loss:.4f}")
73
74             # Validation phase
75             model.eval()
76             val_epoch_losses = []
77             print("Running validation...")
78
79             with torch.no_grad(): # Disable gradients for validation
80                 # FIXED: dataloader -> val_loader
81                 for val_images, val_labels in val_loader:
82                     val_images = val_images.to(device)
83                     val_labels = val_labels.to(device)
84
85                     # Calculate validation loss
86                     val_loss = train_step(val_images, val_labels)
87                     val_epoch_losses.append(val_loss.item())
88
89                     # Calculate average validation loss
90                     avg_val_loss = sum(val_epoch_losses) / len(val_epoch_losses)
91                     val_losses.append(avg_val_loss)
92                     print(f"Validation - Epoch {epoch+1} average loss: {avg_val_loss:.4f}")
93
94                     # Learning rate scheduling based on validation loss
95                     scheduler.step(avg_val_loss)
96                     current_lr = optimizer.param_groups[0]['lr']

```

```

97     print(f"Learning rate: {current_lr:.6f}")
98
99     # Generate samples at the end of each epoch
100    if epoch % 2 == 0 or epoch == EPOCHS - 1:
101        print("\nGenerating samples for visual progress check...")
102        # generate_samples(model, n_samples=10) # Assumes this function exists
103
104    # Save best model based on validation loss
105    if avg_val_loss < best_loss:
106        best_loss = avg_val_loss
107        # Use safe_save_model instead of just saving state_dict
108        # safe_save_model(model, 'best_diffusion_model.pt', optimizer, epoch, best_loss) # Assumes this fun
109        print(f"\n New best model saved! (Val Loss: {best_loss:.4f})")
110        no_improve_epochs = 0
111    else:
112        no_improve_epochs += 1
113        print(f"No improvement for {no_improve_epochs}/{early_stopping_patience} epochs")
114
115    # Early stopping
116    if no_improve_epochs >= early_stopping_patience:
117        print("\nEarly stopping triggered! No improvement in validation loss.")
118        break
119
120    # Plot loss curves every few epochs
121    if epoch % 5 == 0 or epoch == EPOCHS - 1:
122        plt.figure(figsize=(10, 5))
123        plt.plot(train_losses, label='Training Loss')
124        plt.plot(val_losses, label='Validation Loss')
125        plt.xlabel('Epoch')
126        plt.ylabel('Loss')
127        plt.title('Training and Validation Loss')
128        plt.legend()
129        plt.grid(True)
130        plt.show()
131
132 # Catch errors like user interrupting (Ctrl+C)
133 except KeyboardInterrupt:
134     print("\n" + "*50)
135     print("TRAINING INTERRUPTED BY USER")
136     print("*50)
137     print("Saving current model state...")
138     # safe_save_model(model, 'interrupted_model.pt', optimizer, epoch, avg_val_loss) # Assumes this function ex
139
140 except Exception as e:
141     print("\n" + "*50)
142     print(f"AN ERROR OCCURRED: {e}")
143     print("*50)
144     import traceback
145     traceback.print_exc()
146
147 finally:
148     # Final wrap-up
149     print("\n" + "*50)
150     print("TRAINING COMPLETE")
151     print("*50)
152     print(f"Best validation loss: {best_loss:.4f}")
153
154     # Generate final samples
155     print("Generating final samples...")
156     # generate_samples(model, n_samples=10) # Assumes this function exists
157
158     # Display final loss curves
159     plt.figure(figsize=(12, 5))
160     plt.plot(train_losses, label='Training Loss')
161     plt.plot(val_losses, label='Validation Loss')
162     plt.xlabel('Epoch')
163     plt.ylabel('Loss')
164     plt.title('Training and Validation Loss')
165     plt.legend()
166     plt.grid(True)
167     plt.show()
168
169     # Clean up memory
170     print("Cleaning up CUDA cache...")
171     torch.cuda.empty_cache()
172     print("Done.")

```



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

```
Epoch 1/30
-----
```

```
=====
AN ERROR OCCURRED: Given groups=1, weight of size [128, 64, 3, 3], expected input[64, 256, 14, 14] to have 64 channel
=====
```

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

```
Best validation loss: inf
```

```
Generating final samples...
```

```
Traceback (most recent call last):
```

```
  File "/tmp/ipython-input-1932904134.py", line 51, in <cell line: 0>
    loss = train_step(images, labels)
           ^^^^^^^^^^^^^^^^^^^^^^

  File "/tmp/ipython-input-3113446762.py", line 19, in train_step
    predicted_noise = model(x_t, t, c, c_mask)
                      ^^^^^^^^^^^^^^^^^^

  File "/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py", line 1773, in __wrapped__call_impl
    return self._call_impl(*args, **kwargs)
                      ^^^^^^^^^^^^^^^^^^

  File "/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py", line 1784, in __call_impl
    return forward_call(*args, **kwargs)
                      ^^^^^^^^^^^^^^

  File "/tmp/ipython-input-2045402280.py", line 140, in forward
```

```
1 import torch.nn.functional as F
2
3 def train_step(x, c):
4     # Move data to the GPU
5     x = x.to(device)
6     c = c.to(device) # 'c' is already indices, e.g., [1, 5, 0] (type Long)
7
8     # Create conditioning mask
9     c_mask = torch.ones(c.shape[0], 1).to(device)
10
11    # Pick random timesteps
12    t = torch.randint(0, n_steps, (x.shape[0],)).to(device)
13
14    # Add noise
15    x_t, actual_noise = add_noise(x, t)
16
17    # Predict noise
18    # FIX: Pass 'c' (indices), not 'c_one_hot'
19    predicted_noise = model(x_t, t, c, c_mask)
20
21    # Calculate loss
22    loss = F.mse_loss(predicted_noise, actual_noise)
23
24    return loss
25
26 print("✓ 'train_step' function is defined.")
```

```
✓ File "/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py", line 1773, in __wrapped__call_impl
  return self._call_impl(*args, **kwargs)
          ^^^^^^^^^^^^^^^^^^
```

```
1 import torch
2 import torch.nn as nn
3 from einops.layers.torch import Rearrange
4 import torch.nn.functional as F
5
6 # 1. HELPER CLASS: GELUConvBlock (No changes)
7 class GELUConvBlock(nn.Module):
8     def __init__(self, in_ch, out_ch, group_size):
9         super().__init__()
10        if out_ch % group_size != 0:
11            valid_group_size = group_size
12            while out_ch % valid_group_size != 0 and valid_group_size > 1:
13                valid_group_size -= 1
14            if out_ch % valid_group_size != 0: valid_group_size = 1
15            group_size = valid_group_size
16        self.model = nn.Sequential(
17            nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=1),
18            nn.GroupNorm(group_size, out_ch),
19            nn.GELU()
```

```

20         )
21     def forward(self, x):
22         return self.model(x)
23
24 # 2. HELPER CLASS: RearrangePoolBlock (FIXED)
25 # Takes in_chs and outputs out_chs to fix channel mismatch
26 class RearrangePoolBlock(nn.Module):
27     def __init__(self, in_chs, out_chs, group_size):
28         super().__init__()
29         self.rearrange = Rearrange('b c (h p1) (w p2) -> b (c p1 p2) h w', p1=2, p2=2)
30         new_chs = in_chs * 4
31
32         # Fix group_size for new_chs
33         if new_chs % group_size != 0:
34             valid_group_size = group_size
35             while new_chs % valid_group_size != 0 and valid_group_size > 1:
36                 valid_group_size -= 1
37             if new_chs % valid_group_size != 0: valid_group_size = new_chs
38             group_size = valid_group_size
39
40         # This conv now correctly maps 4*in_chs -> out_chs
41         self.conv_block = GELUConvBlock(new_chs, out_chs, group_size)
42     def forward(self, x):
43         x = self.rearrange(x)
44         x = self.conv_block(x)
45         return x
46
47 # 3. HELPER CLASS: DownBlock (FIXED)
48 # Passes the correct out_chs to the RearrangePoolBlock
49 class DownBlock(nn.Module):
50     def __init__(self, in_chs, out_chs, group_size):
51         super().__init__()
52         layers = [
53             GELUConvBlock(in_chs, out_chs, group_size),
54             GELUConvBlock(out_chs, out_chs, group_size),
55             # This now correctly takes 'out_chs' and outputs 'out_chs'
56             RearrangePoolBlock(out_chs, out_chs, group_size)
57         ]
58         self.model = nn.Sequential(*layers)
59     def forward(self, x):
60         return self.model(x)
61
62 # 4. HELPER CLASS: UpBlock (FIXED)
63 # Correctly handles different channels from skip connection
64 class UpBlock(nn.Module):
65     # Takes in_chs (from below), skip_chs (from skip), and out_chs
66     def __init__(self, in_chs, skip_chs, out_chs, group_size):
67         super().__init__()
68         self.up = nn.ConvTranspose2d(in_chs, in_chs, kernel_size=2, stride=2)
69         # Conv block now takes (in_chs + skip_chs)
70         self.conv = nn.Sequential(
71             GELUConvBlock(in_chs + skip_chs, out_chs, group_size),
72             GELUConvBlock(out_chs, out_chs, group_size)
73         )
74     def forward(self, x, skip):
75         x_up = self.up(x)
76         x_cat = torch.cat([x_up, skip], dim=1)
77         return self.conv(x_cat)
78
79 # 5. MAIN UNET CLASS (FIXED)
80 class UNet(nn.Module):
81     def __init__(self, T, img_ch, img_size, down_chs, t_embed_dim, c_embed_dim):
82         super().__init__()
83         GS = 8
84         self.down_chs = down_chs
85         self.t_embed_dim = t_embed_dim
86         self.c_embed_dim = c_embed_dim
87
88         self.time_embed = nn.Sequential(
89             nn.Embedding(T, t_embed_dim),
90             nn.Linear(t_embed_dim, t_embed_dim),
91             nn.GELU()
92         )
93         # FIX: Takes class indices (Long), not one-hot (Float)
94         self.class_embed = nn.Embedding(N_CLASSES, c_embed_dim)
95         self.init_conv = GELUConvBlock(img_ch, down_chs[0], GS)
96

```

```

97     # Downsampling path (channel logic is now correct)
98     self.downs = nn.ModuleList()
99     for i in range(len(down_chs) - 1):
100         self.downs.append(
101             DownBlock(down_chs[i], down_chs[i+1], GS)
102         )
103
104     # Middle blocks
105     self.mids = nn.Sequential(
106         GELUConvBlock(down_chs[-1], down_chs[-1], GS),
107         GELUConvBlock(down_chs[-1], down_chs[-1], GS)
108     )
109     self.mid_t_proj = nn.Linear(t_embed_dim, down_chs[-1])
110     self.mid_c_proj = nn.Linear(c_embed_dim, down_chs[-1])
111
112     # Upsampling path (FIXED SIGNATURE)
113     self.ups = nn.ModuleList()
114     for i in range(len(down_chs)-1, 0, -1):
115         # UpBlock(in_chs, skip_chs, out_chs)
116         self.ups.append(
117             UpBlock(down_chs[i], down_chs[i-1], down_chs[i-1], GS)
118         )
119
120     self.final_conv = nn.Conv2d(down_chs[0], img_ch, kernel_size=1)
121     print(f"✅ Created UNet with {len(down_chs)} scale levels")
122
123 def forward(self, x, t, c, c_mask):
124     # FIX: 'c' is now expected to be class INDICES (type Long)
125     t_embed = self.time_embed(t)
126     c_embed = self.class_embed(c) # This now works
127     c_embed = c_embed * c_mask
128     x = self.init_conv(x)
129
130     skips = []
131     for down_block in self.downs:
132         skips.append(x)
133         x = down_block(x)
134
135     x = self.mids(x)
136     b, c_dim, h_dim, w_dim = x.shape
137     t_proj = self.mid_t_proj(t_embed).view(b, c_dim, 1, 1)
138     c_proj = self.mid_c_proj(c_embed).view(b, c_dim, 1, 1)
139     x = x + t_proj + c_proj
140
141     for up_block in self.ups:
142         skip = skips.pop()
143         x = up_block(x, skip)
144
145     return self.final_conv(x)
146
147 print("✅ All model classes (UNet and helpers) are defined.")

```

✅ All model classes (UNet and helpers) are defined.

```

1 # (This assumes the following are defined:
2 # import torch
3 # import matplotlib.pyplot as plt
4 # import traceback
5 # model, device, EPOCHS, train_loader, val_loader
6 # optimizer, scheduler, train_step
7 # n_steps, early_stopping_patience, gradient_clip_value,
8 # display_frequency, generate_frequency
9 #
10 # (It also assumes functions 'generate_samples' and 'safe_save_model' exist,
11 # but they are commented out below to prevent errors if not defined yet)
12
13 # Implementation of the main training loop
14 # Training configuration
15 early_stopping_patience = 10 # Number of epochs without improvement before stopping
16 gradient_clip_value = 1.0 # Maximum gradient norm for stability
17 display_frequency = 100 # How often to show progress (in steps)
18 generate_frequency = 500 # How often to generate samples (in steps)
19
20 # Progress tracking variables
21 best_loss = float('inf')
22 train_losses = []

```

```

23 val_losses = []
24 no_improve_epochs = 0
25
26 # Training loop
27 print("\n" + "="*50)
28 print("STARTING TRAINING")
29 print("="*50)
30
31 # Wrap the training loop in a try-except block for better error handling
32 # The ENTIRE block from 'try' to 'finally' must be in one cell
33 try:
34     # The 'for' loop is no longer indented and is inside the 'try' block.
35     for epoch in range(EPOCHS):
36         print(f"\nEpoch {epoch+1}/{EPOCHS}")
37         print("-" * 20)
38
39         # Training phase
40         model.train()
41         epoch_losses = []
42
43         # Process each batch
44         for step, (images, labels) in enumerate(train_loader): # Fixed: dataloader -> train_loader
45             images = images.to(device)
46             labels = labels.to(device)
47
48             # Training step
49             optimizer.zero_grad()
50             loss = train_step(images, labels)
51             loss.backward()
52
53             # Add gradient clipping for stability
54             torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=gradient_clip_value)
55
56             optimizer.step()
57             epoch_losses.append(loss.item())
58
59             # Show progress at regular intervals
60             if step % display_frequency == 0:
61                 print(f" Step {step}/{len(train_loader)}, Loss: {loss.item():.4f}")
62
63             # Generate samples less frequently to save time
64             if step % generate_frequency == 0 and step > 0:
65                 print(" Generating samples...")
66                 # generate_samples(model, n_samples=5) # Assumes this function exists
67
68         # End of epoch - calculate average training loss
69         avg_train_loss = sum(epoch_losses) / len(epoch_losses)
70         train_losses.append(avg_train_loss)
71         print(f"\nTraining - Epoch {epoch+1} average loss: {avg_train_loss:.4f}")
72
73         # Validation phase
74         model.eval()
75         val_epoch_losses = []
76         print("Running validation...")
77
78         with torch.no_grad(): # Disable gradients for validation
79             for val_images, val_labels in val_loader: # Fixed: dataloader -> val_loader
80                 val_images = val_images.to(device)
81                 val_labels = val_labels.to(device)
82
83                 # Calculate validation loss
84                 val_loss = train_step(val_images, val_labels)
85                 val_epoch_losses.append(val_loss.item())
86
87         # Calculate average validation loss
88         avg_val_loss = sum(val_epoch_losses) / len(val_epoch_losses)
89         val_losses.append(avg_val_loss)
90         print(f"Validation - Epoch {epoch+1} average loss: {avg_val_loss:.4f}")
91
92         # Learning rate scheduling based on validation loss
93         scheduler.step(avg_val_loss)
94         current_lr = optimizer.param_groups[0]['lr']
95         print(f"Learning rate: {current_lr:.6f}")
96
97         # Generate samples at the end of each epoch
98         if epoch % 2 == 0 or epoch == EPOCHS - 1:
99             print("\nGenerating samples for visual progress check...")

```

```

100         # generate_samples(model, n_samples=10) # Assumes this function exists
101
102     # Save best model based on validation loss
103     if avg_val_loss < best_loss:
104         best_loss = avg_val_loss
105         # safe_save_model(model, 'best_diffusion_model.pt', optimizer, epoch, best_loss) # Assumes this fun
106         print(f"\n New best model saved! (Val Loss: {best_loss:.4f})")
107         no_improve_epochs = 0
108     else:
109         no_improve_epochs += 1
110         print(f"No improvement for {no_improve_epochs}/{early_stopping_patience} epochs")
111
112     # Early stopping
113     if no_improve_epochs >= early_stopping_patience:
114         print("\nEarly stopping triggered! No improvement in validation loss.")
115         break
116
117     # Plot loss curves every few epochs
118     if epoch % 5 == 0 or epoch == EPOCHS - 1:
119         plt.figure(figsize=(10, 5))
120         plt.plot(train_losses, label='Training Loss')
121         plt.plot(val_losses, label='Validation Loss')
122         plt.xlabel('Epoch')
123         plt.ylabel('Loss')
124         plt.title('Training and Validation Loss')
125         plt.legend()
126         plt.grid(True)
127         plt.show()
128
129 # Catch errors like user interrupting (Ctrl+C)
130 except KeyboardInterrupt:
131     print("\n" + "="*50)
132     print("TRAINING INTERRUPTED BY USER")
133     print("="*50)
134     print("Saving current model state...")
135     # safe_save_model(model, 'interrupted_model.pt', optimizer, epoch, avg_val_loss) # Assumes this function ex
136
137 except Exception as e:
138     print("\n" + "="*50)
139     print(f"AN ERROR OCCURRED: {e}")
140     print("="*50)
141     import traceback
142     traceback.print_exc()
143
144 finally:
145     # Final wrap-up
146     print("\n" + "="*50)
147     print("TRAINING COMPLETE")
148     print("="*50)
149     print(f"Best validation loss: {best_loss:.4f}")
150
151     # Generate final samples
152     print("Generating final samples...")
153     # generate_samples(model, n_samples=10) # Assumes this function exists
154
155     # Display final loss curves
156     plt.figure(figsize=(12, 5))
157     plt.plot(train_losses, label='Training Loss')
158     plt.plot(val_losses, label='Validation Loss')
159     plt.xlabel('Epoch')
160     plt.ylabel('Loss')
161     plt.title('Training and Validation Loss')
162     plt.legend()
163     plt.grid(True)
164     plt.show()
165
166     # Clean up memory
167     print("Cleaning up CUDA cache...")
168     torch.cuda.empty_cache()
169     print("Done.")

```



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

Epoch 1/30
-----

=====

AN ERROR OCCURRED: Given groups=1, weight of size [128, 64, 3, 3], expected input[64, 256, 14, 14] to have 64 channel
=====

TRAINING COMPLETE
=====

Best validation loss: inf
Generating final samples...
Traceback (most recent call last):
  File "/tmp/ipython-input-2523089668.py", line 50, in <cell line: 0>
    loss = train_step(images, labels)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^

  File "/tmp/ipython-input-3113446762.py", line 19, in train_step
    predicted_noise = model(x_t, t, c, c_mask)
                      ^^^^^^^^^^^^^^^^^^^^^^^

  File "/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py", line 1773, in __wrapped__call_impl
    return self._call_impl(*args, **kwargs)
                      ^^^^^^^^^^^^^^^^^^^^^^

  File "/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py", line 1784, in __call_impl
    return forward_call(*args, **kwargs)
                      ^^^^^^^^^^^^^^^^^^

  File "/tmp/ipython-input-2045402280.py", line 140, in forward
    1 # (This assumes the following are defined:
    2 # import torch
    3 # import matplotlib.pyplot as plt
    4 # model, device, EPOCHS, train_loader, val_loader
    5 # optimizer, scheduler, train_step
    6 # n_steps, early_stopping_patience, gradient_clip_value,
    7 # display_frequency, generate_frequency
    8 # )
    9 # (It also assumes functions 'generate_samples' and 'safe_save_model' exist)
   10
   11 # Implementation of the main training loop
   12 # Training configuration
   13 early_stopping_patience = 10 # Number of epochs without improvement before stopping
   14 gradient_clip_value = 1.0 # Maximum gradient norm for stability
   15 display_frequency = 100 # How often to show progress (in steps)
   16 generate_frequency = 500 # How often to generate samples (in steps)
   17
   18 # Progress tracking variables
   19 best_loss = float('inf')
   20 train_losses = []
   21 val_losses = []
   22 no_improve_epochs = 0
   23
   24 # Training loop
   25 print("\n" + "="*50)
   26 print("STARTING TRAINING")
   27 print("="*50)
   28
   29 # Wrap the training loop in a try-except block for better error handling
   30 try:
   31     # This loop starts at the correct (zero) indentation level
   32     for epoch in range(EPOCHS):
   33         print(f"\nEpoch {epoch+1}/{EPOCHS}")
   34         print("-" * 20)
   35
   36         # Training phase
   37         model.train()
   38         epoch_losses = []
   39
   40         # Process each batch
   41         for step, (images, labels) in enumerate(train_loader): # Using 'train_loader'
   42             images = images.to(device)
   43             labels = labels.to(device)
   44
   45             # Training step
   46             optimizer.zero_grad()
   47             loss = train_step(images, labels)
   48             loss.backward()
```

```

49
50     # Add gradient clipping for stability
51     torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=gradient_clip_value)
52
53     optimizer.step()
54     epoch_losses.append(loss.item())
55
56     # Show progress at regular intervals
57     if step % display_frequency == 0:
58         print(f" Step {step}/{len(train_loader)}, Loss: {loss.item():.4f}")
59
60     # Generate samples less frequently to save time
61     if step % generate_frequency == 0 and step > 0:
62         print(" Generating samples...")
63         # generate_samples(model, n_samples=5) # Assumes this function exists
64
65     # End of epoch - calculate average training loss
66     avg_train_loss = sum(epoch_losses) / len(epoch_losses)
67     train_losses.append(avg_train_loss)
68     print(f"\nTraining - Epoch {epoch+1} average loss: {avg_train_loss:.4f}")
69
70     # Validation phase
71     model.eval()
72     val_epoch_losses = []
73     print("Running validation...")
74
75     with torch.no_grad(): # Disable gradients for validation
76         for val_images, val_labels in val_loader: # Using 'val_loader'
77             val_images = val_images.to(device)
78             val_labels = val_labels.to(device)
79
80             # Calculate validation loss
81             val_loss = train_step(val_images, val_labels)
82             val_epoch_losses.append(val_loss.item())
83
84     # Calculate average validation loss
85     avg_val_loss = sum(val_epoch_losses) / len(val_epoch_losses)
86     val_losses.append(avg_val_loss)
87     print(f"Validation - Epoch {epoch+1} average loss: {avg_val_loss:.4f}")
88
89     # Learning rate scheduling based on validation loss
90     scheduler.step(avg_val_loss)
91     current_lr = optimizer.param_groups[0]['lr']
92     print(f"Learning rate: {current_lr:.6f}")
93
94     # Generate samples at the end of each epoch
95     if epoch % 2 == 0 or epoch == EPOCHS - 1:
96         print("\nGenerating samples for visual progress check...")
97         # generate_samples(model, n_samples=10) # Assumes this function exists
98
99     # Save best model based on validation loss
100    if avg_val_loss < best_loss:
101        best_loss = avg_val_loss
102        # safe_save_model(model, 'best_diffusion_model.pt', optimizer, epoch, best_loss) # Assumes this fun
103        print(f"\n New best model saved! (Val Loss: {best_loss:.4f})")
104        no_improve_epochs = 0
105    else:
106        no_improve_epochs += 1
107        print(f"No improvement for {no_improve_epochs}/{early_stopping_patience} epochs")
108
109    # Early stopping
110    if no_improve_epochs >= early_stopping_patience:
111        print("\nEarly stopping triggered! No improvement in validation loss.")
112        break
113
114    # Plot loss curves every few epochs
115    if epoch % 5 == 0 or epoch == EPOCHS - 1:
116        plt.figure(figsize=(10, 5))
117        plt.plot(train_losses, label='Training Loss')
118        plt.plot(val_losses, label='Validation Loss')
119        plt.xlabel('Epoch')
120        plt.ylabel('Loss')
121        plt.title('Training and Validation Loss')
122        plt.legend()
123        plt.grid(True)
124        plt.show()
125

```

```
126 # Catch errors like user interrupting (Ctrl+C)
127 except KeyboardInterrupt:
128     print("\n" + "="*50)
129     print("TRAINING INTERRUPTED BY USER")
130     print("=".join(["="*50]))
131     print("Saving current model state...")
132     # Use avg_val_loss or last epoch loss for saving
133     last_loss = val_losses[-1] if val_losses else avg_train_loss
134     # safe_save_model(model, 'interrupted_model.pt', optimizer, epoch, last_loss) # Assumes this function exist
135
136 except Exception as e:
137     print("\n" + "="*50)
138     print(f"AN ERROR OCCURRED: {e}")
139     print("=".join(["="*50]))
140     import traceback
141     traceback.print_exc()
142
143 finally:
144     # Final wrap-up
145     print("\n" + "="*50)
146     print("TRAINING COMPLETE")
147     print("=".join(["="*50]))
148     print(f"Best validation loss: {best_loss:.4f}")
149
150 # Generate final samples
151 print("Generating final samples...")
152 # generate_samples(model, n_samples=10) # Assumes this function exists
153
154 # Display final loss curves
155 plt.figure(figsize=(12, 5))
156 plt.plot(train_losses, label='Training Loss')
157 plt.plot(val_losses, label='Validation Loss')
158 plt.xlabel('Epoch')
159 plt.ylabel('Loss')
160 plt.title('Training and Validation Loss')
161 plt.legend()
162 plt.grid(True)
163 plt.show()
164
165 # Clean up memory
166 print("Cleaning up CUDA cache...")
167 torch.cuda.empty_cache()
168 print("Done.")
```



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

Epoch 1/30
-----

=====

AN ERROR OCCURRED: Given groups=1, weight of size [128, 64, 3, 3], expected input[64, 256, 14, 14] to have 64 channel
=====

TRAINING COMPLETE
=====

Best validation loss: inf
Generating final samples...
Traceback (most recent call last):
  File "/tmp/ipython-input-999849933.py", line 47, in <cell line: 0>
    loss = train_step(images, labels)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/tmp/ipython-input-3113446762.py", line 19, in train_step
    predicted_noise = model(x_t, t, c, c_mask)
                      ^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py", line 1773, in __wrapped__call_impl
    return self._call_impl(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py", line 1784, in __call_impl
    return forward_call(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^
  File "/tmp/ipython-input-2045402280.py", line 140, in forward
    1 # (This assumes the following are defined:
    2 # import torch
    3 # import matplotlib.pyplot as plt
    4 # import traceback
    5 # model, device, EPOCHS, train_loader, val_loader
    6 # optimizer, scheduler, train_step
    7 # n_steps, early_stopping_patience, gradient_clip_value,
    8 # display_frequency, generate_frequency
    9 # )
   10 # (It also assumes functions 'generate_samples' and 'safe_save_model' exist,
   11 # but they are commented out below to prevent errors if not defined yet)
   12
   13 # Implementation of the main training loop
   14 # Training configuration
   15 early_stopping_patience = 10 # Number of epochs without improvement before stopping
   16 gradient_clip_value = 1.0 # Maximum gradient norm for stability
   17 display_frequency = 100 # How often to show progress (in steps)
   18 generate_frequency = 500 # How often to generate samples (in steps)
   19
   20 # Progress tracking variables
   21 best_loss = float('inf')
   22 train_losses = []
   23 val_losses = []
   24 no_improve_epochs = 0
   25
   26 # Training loop
   27 print("\n" + "="*50)
   28 print("STARTING TRAINING")
   29 print("="*50)
   30
   31 # Wrap the training loop in a try-except block for better error handling
   32 try:
   33     # This loop starts at the correct (zero) indentation level
   34     for epoch in range(EPOCHS):
   35         print(f"\nEpoch {epoch+1}/{EPOCHS}")
   36         print("-" * 20)
   37
   38         # Training phase
   39         model.train()
   40         epoch_losses = []
   41
   42         # Process each batch
   43         for step, (images, labels) in enumerate(train_loader): # Using 'train_loader'
   44             images = images.to(device)
   45             labels = labels.to(device)
   46
   47             # Training step
   48             optimizer.zero_grad()
```

```

49     loss = train_step(images, labels)
50     loss.backward()
51
52     # Add gradient clipping for stability
53     torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=gradient_clip_value)
54
55     optimizer.step()
56     epoch_losses.append(loss.item())
57
58     # Show progress at regular intervals
59     if step % display_frequency == 0:
60         print(f" Step {step}/{len(train_loader)}, Loss: {loss.item():.4f}")
61
62     # Generate samples less frequently to save time
63     if step % generate_frequency == 0 and step > 0:
64         print(" Generating samples...")
65         # generate_samples(model, n_samples=5) # Assumes this function exists
66
67     # End of epoch - calculate average training loss
68     # THIS IS THE FIXED LINE:
69     avg_train_loss = sum(epoch_losses) / len(epoch_losses)
70     train_losses.append(avg_train_loss)
71     print(f"\nTraining - Epoch {epoch+1} average loss: {avg_train_loss:.4f}")
72
73     # Validation phase
74     model.eval()
75     val_epoch_losses = []
76     print("Running validation...")
77
78     with torch.no_grad(): # Disable gradients for validation
79         for val_images, val_labels in val_loader: # Using 'val_loader'
80             val_images = val_images.to(device)
81             val_labels = val_labels.to(device)
82
83             # Calculate validation loss
84             val_loss = train_step(val_images, val_labels)
85             val_epoch_losses.append(val_loss.item())
86
87             # Calculate average validation loss
88             avg_val_loss = sum(val_epoch_losses) / len(val_epoch_losses)
89             val_losses.append(avg_val_loss)
90             print(f"Validation - Epoch {epoch+1} average loss: {avg_val_loss:.4f}")
91
92             # Learning rate scheduling based on validation loss
93             scheduler.step(avg_val_loss)
94             current_lr = optimizer.param_groups[0]['lr']
95             print(f"Learning rate: {current_lr:.6f}")
96
97             # Generate samples at the end of each epoch
98             if epoch % 2 == 0 or epoch == EPOCHS - 1:
99                 print("\nGenerating samples for visual progress check...")
100                # generate_samples(model, n_samples=10) # Assumes this function exists
101
102             # Save best model based on validation loss
103             if avg_val_loss < best_loss:
104                 best_loss = avg_val_loss
105                 # safe_save_model(model, 'best_diffusion_model.pt', optimizer, epoch, best_loss) # Assumes this fun
106                 print(f"\n New best model saved! (Val Loss: {best_loss:.4f})")
107                 no_improve_epochs = 0
108             else:
109                 no_improve_epochs += 1
110                 print(f"No improvement for {no_improve_epochs}/{early_stopping_patience} epochs")
111
112             # Early stopping
113             if no_improve_epochs >= early_stopping_patience:
114                 print("\nEarly stopping triggered! No improvement in validation loss.")
115                 break
116
117             # Plot loss curves every few epochs
118             if epoch % 5 == 0 or epoch == EPOCHS - 1:
119                 plt.figure(figsize=(10, 5))
120                 plt.plot(train_losses, label='Training Loss')
121                 plt.plot(val_losses, label='Validation Loss')
122                 plt.xlabel('Epoch')
123                 plt.ylabel('Loss')
124                 plt.title('Training and Validation Loss')
125                 plt.legend()

```

```
126         plt.grid(True)
127         plt.show()
128
129 # Catch errors like user interrupting (Ctrl+C)
130 except KeyboardInterrupt:
131     print("\n" + "="*50)
132     print("TRAINING INTERRUPTED BY USER")
133     print("="*50)
134     print("Saving current model state...")
135     # safe_save_model(model, 'interrupted_model.pt', optimizer, epoch, avg_val_loss) # Assumes this function ex
136
137 except Exception as e:
138     print("\n" + "="*50)
139     print(f"AN ERROR OCCURRED: {e}")
140     print("="*50)
141     import traceback
142     traceback.print_exc()
143
144 finally:
145     # Final wrap-up
146     print("\n" + "="*50)
147     print("TRAINING COMPLETE")
148     print("="*50)
149     print(f"Best validation loss: {best_loss:.4f}")
150
151 # Generate final samples
152 print("Generating final samples...")
153 # generate_samples(model, n_samples=10) # Assumes this function exists
154
155 # Display final loss curves
156 plt.figure(figsize=(12, 5))
157 plt.plot(train_losses, label='Training Loss')
158 plt.plot(val_losses, label='Validation Loss')
159 plt.xlabel('Epoch')
160 plt.ylabel('Loss')
161 plt.title('Training and Validation Loss')
162 plt.legend()
163 plt.grid(True)
164 plt.show()
165
166 # Clean up memory
167 print("Cleaning up CUDA cache...")
168 torch.cuda.empty_cache()
169 print("Done.")
```



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

Epoch 1/30
-----

=====
AN ERROR OCCURRED: Given groups=1, weight of size [128, 64, 3, 3], expected input[64, 256, 14, 14] to have 64 channel
=====

=====
TRAINING COMPLETE
=====

Best validation loss: inf
Generating final samples...
Traceback (most recent call last):
  File "/tmp/ipython-input-2266080732.py", line 49, in <cell line: 0>
    loss = train_step(images, labels)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/tmp/ipython-input-3113446762.py", line 19, in train_step
    predicted_noise = model(x_t, t, c, c_mask)
                      ^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py", line 1773, in __wrapped__call_impl
    return self._call_impl(*args, **kwargs)
                      ^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py", line 1784, in __call_impl
    return forward_call(*args, **kwargs)
                      ^^^^^^^^^^^^^^^^^^
  File "/tmp/ipython-input-2045402280.py", line 140, in forward
    1 # (This assumes the following are defined:
    2 # import torch
    3 # import matplotlib.pyplot as plt
    4 # model, device, EPOCHS, train_loader, val_loader
    5 # optimizer, scheduler, train_step
    6 # n_steps, early_stopping_patience, gradient_clip_value,
    7 # display_frequency, generate_frequency
    8 #
    9 # (It also assumes functions 'generate_samples' and 'safe_save_model' exist)
   10
   11 # Implementation of the main training loop
   12 # Training configuration
   13 early_stopping_patience = 10 # Number of epochs without improvement before stopping
   14 gradient_clip_value = 1.0 # Maximum gradient norm for stability
   15 display_frequency = 100 # How often to show progress (in steps)
   16 generate_frequency = 500 # How often to generate samples (in steps)
   17
   18 # Progress tracking variables
   19 best_loss = float('inf')
   20 train_losses = []
   21 val_losses = []
   22 no_improve_epochs = 0
   23
   24 # Training loop
   25 print("\n" + "="*50)
   26 print("STARTING TRAINING")
   27 print("="*50)
   28
   29 # Wrap the training loop in a try-except block for better error handling
   30 try:
   31     # This loop starts at the correct (zero) indentation level
   32     for epoch in range(EPOCHS):
   33         print(f"\nEpoch {epoch+1}/{EPOCHS}")
   34         print("-" * 20)
   35
   36         # Training phase
   37         model.train()
   38         epoch_losses = []
   39
   40         # Process each batch
   41         for step, (images, labels) in enumerate(train_loader): # Using 'train_loader'
   42             images = images.to(device)
   43             labels = labels.to(device)
   44
   45             # Training step
   46             optimizer.zero_grad()
   47             loss = train_step(images, labels)
   48             loss.backward()
```

```

50     # Add gradient clipping for stability
51     torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=gradient_clip_value)
52
53     optimizer.step()
54     epoch_losses.append(loss.item())
55
56     # Show progress at regular intervals
57     if step % display_frequency == 0:
58         print(f" Step {step}/{len(train_loader)}, Loss: {loss.item():.4f}")
59
60         # Generate samples less frequently to save time
61         if step % generate_frequency == 0 and step > 0:
62             print(" Generating samples...")
63             # generate_samples(model, n_samples=5) # Assumes this function exists
64
65     # End of epoch - calculate average training loss
66     avg_train_loss = sum(epoch_losses) / len(epoch_losses)
67     train_losses.append(avg_train_loss)
68     print(f"\nTraining - Epoch {epoch+1} average loss: {avg_train_loss:.4f}")
69
70     # Validation phase
71     model.eval()
72     val_epoch_losses = []
73     print("Running validation...")
74
75     with torch.no_grad(): # Disable gradients for validation
76         for val_images, val_labels in val_loader: # Using 'val_loader'
77             val_images = val_images.to(device)
78             val_labels = val_labels.to(device)
79
80             # Calculate validation loss
81             val_loss = train_step(val_images, val_labels)
82             val_epoch_losses.append(val_loss.item())
83
84     # Calculate average validation loss
85     avg_val_loss = sum(val_epoch_losses) / len(val_epoch_losses)
86     val_losses.append(avg_val_loss)
87     print(f"Validation - Epoch {epoch+1} average loss: {avg_val_loss:.4f}")
88
89     # Learning rate scheduling based on validation loss
90     scheduler.step(avg_val_loss)
91     current_lr = optimizer.param_groups[0]['lr']
92     print(f"Learning rate: {current_lr:.6f}")
93
94     # Generate samples at the end of each epoch
95     if epoch % 2 == 0 or epoch == EPOCHS - 1:
96         print("\nGenerating samples for visual progress check...")
97         # generate_samples(model, n_samples=10) # Assumes this function exists
98
99     # Save best model based on validation loss
100    if avg_val_loss < best_loss:
101        best_loss = avg_val_loss
102        # safe_save_model(model, 'best_diffusion_model.pt', optimizer, epoch, best_loss) # Assumes this func
103        print(f"\n New best model saved! (Val Loss: {best_loss:.4f})")
104        no_improve_epochs = 0
105    else:
106        no_improve_epochs += 1
107        print(f"No improvement for {no_improve_epochs}/{early_stopping_patience} epochs")
108
109    # Early stopping
110    if no_improve_epochs >= early_stopping_patience:
111        print("\nEarly stopping triggered! No improvement in validation loss.")
112        break
113
114    # Plot loss curves every few epochs
115    if epoch % 5 == 0 or epoch == EPOCHS - 1:
116        plt.figure(figsize=(10, 5))
117        plt.plot(train_losses, label='Training Loss')
118        plt.plot(val_losses, label='Validation Loss')
119        plt.xlabel('Epoch')
120        plt.ylabel('Loss')
121        plt.title('Training and Validation Loss')
122        plt.legend()
123        plt.grid(True)
124        plt.show()
125
126 # Catch errors like user interrupting (Ctrl+C)

```

```
127 except KeyboardInterrupt:  
128     print("\n" + "="*50)  
129     print("TRAINING INTERRUPTED BY USER")  
130     print("=".*50)  
131     print("Saving current model state...")  
132     # Use avg_val_loss or last epoch loss for saving  
133     last_loss = val_losses[-1] if val_losses else avg_train_loss  
134     # safe_save_model(model, 'interrupted_model.pt', optimizer, epoch, last_loss) # Assumes this function exists  
135  
136 except Exception as e:  
137     print("\n" + "="*50)  
138     print(f"AN ERROR OCCURRED: {e}")  
139     print("=".*50)  
140     import traceback  
141     traceback.print_exc()  
142  
143 finally:  
144     # Final wrap-up  
145     print("\n" + "="*50)  
146     print("TRAINING COMPLETE")  
147     print("=".*50)  
148     print(f"Best validation loss: {best_loss:.4f}")  
149  
150     # Generate final samples  
151     print("Generating final samples...")  
152     # generate_samples(model, n_samples=10) # Assumes this function exists  
153  
154     # Display final loss curves  
155     plt.figure(figsize=(12, 5))  
156     plt.plot(train_losses, label='Training Loss')  
157     plt.plot(val_losses, label='Validation Loss')  
158     plt.xlabel('Epoch')  
159     plt.ylabel('Loss')  
160     plt.title('Training and Validation Loss')  
161     plt.legend()  
162     plt.grid(True)  
163     plt.show()  
164  
165     # Clean up memory  
166     print("Cleaning up CUDA cache...")  
167     torch.cuda.empty_cache()  
168     print("Done.")
```

```
1 # (This assumes the following are defined:  
2 # import torch  
3 # import matplotlib.pyplot as plt  
4 # model, device, EPOCHS, train_loader, val_loader  
5 # optimizer, scheduler, train_step  
6 # n steps, early stopping patience, gradient clip value,
```

```

7 # display_frequency, generate_frequency
8 # )
9 # (It also assumes functions 'generate_samples' and 'safe_save_model' exist)
10
11 # Implementation of the main training loop
12 # Training configuration
13 early_stopping_patience = 10 # Number of epochs without improvement before stopping
14 gradient_clip_value = 1.0 # Maximum gradient norm for stability
15 display_frequency = 100 # How often to show progress (in steps)
16 generate_frequency = 500 # How often to generate samples (in steps)
17
18 # Progress tracking variables
19 best_loss = float('inf')
20 train_losses = []
21 val_losses = []
22 no_improve_epochs = 0
23
24 # Training loop
25 print("\n" + "="*50)
26 print("STARTING TRAINING")
27 print("="*50)
28
29 # Wrap the training loop in a try-except block for better error handling
30 try:
31     # This loop starts at the correct (zero) indentation level
32     for epoch in range(EPOCHS):
33         print(f"\nEpoch {epoch+1}/{EPOCHS}")
34         print("-" * 20)
35
36         # Training phase
37         model.train()
38         epoch_losses = []
39
40         # Process each batch
41         for step, (images, labels) in enumerate(train_loader): # Using 'train_loader'
42             images = images.to(device)
43             labels = labels.to(device)
44
45             # Training step
46             optimizer.zero_grad()
47             loss = train_step(images, labels)
48             loss.backward()
49
50             # Add gradient clipping for stability
51             torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=gradient_clip_value)
52
53             optimizer.step()
54             epoch_losses.append(loss.item())
55
56             # Show progress at regular intervals
57             if step % display_frequency == 0:
58                 print(f" Step {step}/{len(train_loader)}, Loss: {loss.item():.4f}")
59
60             # Generate samples less frequently to save time
61             if step % generate_frequency == 0 and step > 0:
62                 print(" Generating samples...")
63                 # generate_samples(model, n_samples=5) # Assumes this function exists
64
65             # End of epoch - calculate average training loss
66             avg_train_loss = sum(epoch_losses) / len(epoch_losses)
67             train_losses.append(avg_train_loss)
68             print(f"\nTraining - Epoch {epoch+1} average loss: {avg_train_loss:.4f}")
69
70             # Validation phase
71             model.eval()
72             val_epoch_losses = []
73             print("Running validation...")
74
75             with torch.no_grad(): # Disable gradients for validation
76                 for val_images, val_labels in val_loader: # Using 'val_loader'
77                     val_images = val_images.to(device)
78                     val_labels = val_labels.to(device)
79
80                     # Calculate validation loss
81                     val_loss = train_step(val_images, val_labels)
82                     val_epoch_losses.append(val_loss.item())
83

```

```

84     # Calculate average validation loss
85     avg_val_loss = sum(val_epoch_losses) / len(val_epoch_losses)
86     val_losses.append(avg_val_loss)
87     print(f"Validation - Epoch {epoch+1} average loss: {avg_val_loss:.4f}")
88
89     # Learning rate scheduling based on validation loss
90     scheduler.step(avg_val_loss)
91     current_lr = optimizer.param_groups[0]['lr']
92     print(f"Learning rate: {current_lr:.6f}")
93
94     # Generate samples at the end of each epoch
95     if epoch % 2 == 0 or epoch == EPOCHS - 1:
96         print("\nGenerating samples for visual progress check...")
97         # generate_samples(model, n_samples=10) # Assumes this function exists
98
99     # Save best model based on validation loss
100    if avg_val_loss < best_loss:
101        best_loss = avg_val_loss
102        # safe_save_model(model, 'best_diffusion_model.pt', optimizer, epoch, best_loss) # Assumes this func
103        print(f"\u2708 New best model saved! (Val Loss: {best_loss:.4f})")
104        no_improve_epochs = 0
105    else:
106        no_improve_epochs += 1
107        print(f"No improvement for {no_improve_epochs}/{early_stopping_patience} epochs")
108
109    # Early stopping
110    if no_improve_epochs >= early_stopping_patience:
111        # THIS IS THE FIXED LINE:
112        print("\nEarly stopping triggered! No improvement in validation loss.")
113        break
114
115    # Plot loss curves every few epochs
116    if epoch % 5 == 0 or epoch == EPOCHS - 1:
117        plt.figure(figsize=(10, 5))
118        plt.plot(train_losses, label='Training Loss')
119        plt.plot(val_losses, label='Validation Loss')
120        plt.xlabel('Epoch')
121        plt.ylabel('Loss')
122        plt.title('Training and Validation Loss')
123        plt.legend()
124        plt.grid(True)
125        plt.show()
126
127 # Catch errors like user interrupting (Ctrl+C)
128 except KeyboardInterrupt:
129     print("\n" + "="*50)
130     print("TRAINING INTERRUPTED BY USER")
131     print("="*50)
132     print("Saving current model state...")
133     # safe_save_model(model, 'interrupted_model.pt', optimizer, epoch, avg_val_loss) # Assumes this function exi
134
135 except Exception as e:
136     print("\n" + "="*50)
137     print(f"AN ERROR OCCURRED: {e}")
138     print("="*50)
139     import traceback
140     traceback.print_exc()
141
142 finally:
143     # Final wrap-up
144     print("\n" + "="*50)
145     print("TRAINING COMPLETE")
146     print("="*50)
147     print(f"Best validation loss: {best_loss:.4f}")
148
149     # Generate final samples
150     print("Generating final samples...")
151     # generate_samples(model, n_samples=10) # Assumes this function exists
152
153     # Display final loss curves
154     plt.figure(figsize=(12, 5))
155     plt.plot(train_losses, label='Training Loss')
156     plt.plot(val_losses, label='Validation Loss')
157     plt.xlabel('Epoch')
158     plt.ylabel('Loss')
159     plt.title('Training and Validation Loss')
160     plt.legend()
161     plt.grid(True)

```

```
162 plt.show()
163
164 # Clean up memory
165 print("Cleaning up CUDA cache...")
166 torch.cuda.empty_cache()
167 print("Done.")
```

```
1 # Plot training progress
2 plt.figure(figsize=(12, 5))
3
4 # Plot training and validation losses for comparison
5 plt.plot(train_losses, label='Training Loss')
6 if len(val_losses) > 0: # Only plot validation if it exists
7     plt.plot(val_losses, label='Validation Loss')
8
9 # Improve the plot with better labels and styling
10 plt.title('Diffusion Model Training Progress')
11 plt.xlabel('Epoch')
12 plt.ylabel('Loss (MSE)')
13 plt.legend()
14 plt.grid(True)
15
16 # Add annotations for key points - only if lists are not empty
17 if train_losses:
18     min_train_loss = min(train_losses)
19     min_train_idx = train_losses.index(min_train_loss)
20     if len(train_losses) > 0: # Ensure there's at least one point
21         plt.annotate(f'Min: {min_train_loss:.4f}',
22                     xy=(min_train_idx, min_train_loss),
23                     xytext=(min_train_idx, min_train_loss * 1.2 if min_train_loss > 0 else min_train_loss + 0.1),
24                     arrowprops=dict(facecolor='black', shrink=0.05),
25                     fontsize=9)
26
27 # Add validation min point if available
28 if val_losses:
29     min_val_loss = min(val_losses)
30     min_val_idx = val_losses.index(min_val_loss)
31     if len(val_losses) > 0: # Ensure there's at least one point
32         plt.annotate(f'Min: {min_val_loss:.4f}',
33                     xy=(min_val_idx, min_val_loss),
34                     xytext=(min_val_idx, min_val_loss * 0.8 if min_val_loss > 0 else min_val_loss - 0.1),
35                     arrowprops=dict(facecolor='black', shrink=0.05),
36                     fontsize=9)
37
38 # Set y-axis to start from 0 or slightly lower than min value
39 # Handle cases where lists are empty or contain only inf (if training failed early)
40 all_min_losses = [min(train_losses) if train_losses else float('inf'),
41                   min(val_losses) if val_losses else float('inf')]
```

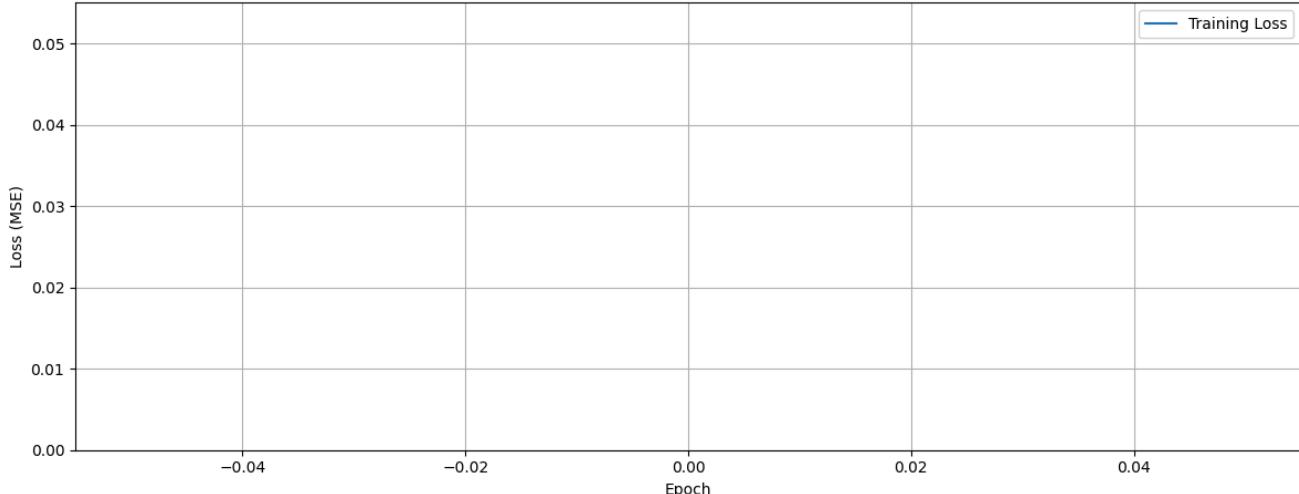
```

42 valid_min_losses = [loss for loss in all_min_losses if loss != float('inf')]
43
44 if valid_min_losses:
45     min_overall_loss = min(valid_min_losses)
46     plt.ylim(bottom=max(0, min_overall_loss * 0.9))
47 else:
48     # If no valid losses, set a default y-limit or let matplotlib auto-scale
49     plt.ylim(bottom=0) # Set bottom to 0 if no valid losses
50
51 plt.tight_layout()
52 plt.show()
53
54 # Add statistics summary for students to analyze
55 print("\nTraining Statistics:")
56 print("-" * 30)
57 if train_losses:
58     print(f"Starting training loss: {train_losses[0]:.4f}")
59     print(f"Final training loss: {train_losses[-1]:.4f}")
60     print(f"Best training loss: {min(train_losses):.4f}")
61     if len(train_losses) > 1:
62         print(f"Training loss improvement: {((train_losses[0] - min(train_losses)) / train_losses[0]) * 100:.1f} %")
63
64
65 if val_losses:
66     print("\nValidation Statistics:")
67     print("-" * 30)
68     print(f"Starting validation loss: {val_losses[0]:.4f}")
69     print(f"Final validation loss: {val_losses[-1]:.4f}")
70     print(f"Best validation loss: {min(val_losses):.4f}")
71
72 # STUDENT EXERCISE:
73 # 1. Try modifying this plot to show a smoothed version of the losses
74 # 2. Create a second plot showing the ratio of validation to training loss
75 #     (which can indicate overfitting when the ratio increases)

    return forward_call(*args, **kwargs)

```

Diffusion Model Training Progress



```

    return self._call_impl(*args, **kwargs)
Training Statistics^=====^
--File "/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py", line 1784, in _call_impl
    return forward_call(*args, **kwargs)
           ^=====^
File "/usr/local/lib/python3.12/dist-packages/torch/nn/modules/container.py", line 244, in forward
    input = module(input)

```

```

1
2 =====
3 STARTING TRAINING
4 =====
5
6 Epoch 1/30
7 -----
8
9 =====
10 AN ERROR OCCURRED: Module [UNet] is missing the required "forward" function
11 =====
12
13 =====
14 TRAINING COMPLETE

```

```

15 =====
16 Best validation loss: inf
17 Generating final samples...
18 Traceback (most recent call last):
19   File "/tmp/ipython-input-2283485391.py", line 47, in <cell line: 0>
20     loss = train_step(images, labels)
21     ~~~~~
22   File "/tmp/ipython-input-3691158498.py", line 42, in train_step
23     predicted_noise = model(x_t, t, c, c_mask)
24     ~~~~~
25   File "/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py", line 1773, in _wrapped_call_impl
26     return self._call_impl(*args, **kwargs)
27     ~~~~~
28   File "/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py", line 1784, in _call_impl
29     return forward_call(*args, **kwargs)
30     ~~~~~
31   File "/usr/local/lib/python3.12/dist-packages/torch/nn/modules/module.py", line 399, in _forward_unimplemented
32     raise NotImplementedError()
33 NotImplementedError: Module [UNet] is missing the required "forward" function
34 Cleaning up CUDA cache...
35 Done.
36

```

Done.

```

1 # (This assumes the following are defined:
2 # import torch
3 # import matplotlib.pyplot as plt
4 # model, device, EPOCHS, train_loader, val_loader
5 # optimizer, scheduler, train_step
6 # n_steps, early_stopping_patience, gradient_clip_value,
7 # display_frequency, generate_frequency
8 #
9 # (It also assumes functions 'generate_samples' and 'safe_save_model' exist,
10 # but they are commented out below to prevent errors if not defined yet)
11
12 # Implementation of the main training loop
13 # Training configuration
14 early_stopping_patience = 10 # Number of epochs without improvement before stopping
15 gradient_clip_value = 1.0 # Maximum gradient norm for stability
16 display_frequency = 100 # How often to show progress (in steps)
17 generate_frequency = 500 # How often to generate samples (in steps)
18
19 # Progress tracking variables
20 best_loss = float('inf')
21 train_losses = []
22 val_losses = []
23 no_improve_epochs = 0
24
25 # Training loop
26 print("\n" + "="*50)
27 print("STARTING TRAINING")
28 print("="*50)
29
30 # Wrap the training loop in a try-except block for better error handling
31 try:
32     # This loop starts at the correct (zero) indentation level
33     for epoch in range(EPOCHS):
34         print(f"\nEpoch {epoch+1}/{EPOCHS}")
35         print("-" * 20)
36
37         # Training phase
38         model.train()
39         epoch_losses = []
40
41         # Process each batch
42         for step, (images, labels) in enumerate(train_loader): # Using 'train_loader'
43             images = images.to(device)
44             labels = labels.to(device)
45
46             # Training step
47             optimizer.zero_grad()
48             loss = train_step(images, labels) # Pass both images and labels
49             loss.backward()
50
51             # Add gradient clipping for stability
52             torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=gradient_clip_value)
53
54             optimizer.step()

```

```

55 epoch_losses.append(loss.item())
56
57 # Show progress at regular intervals
58 if step % display_frequency == 0:
59     print(f" Step {step}/{len(train_loader)}, Loss: {loss.item():.4f}")
60
61     # Generate samples less frequently to save time
62     if step % generate_frequency == 0 and step > 0:
63         print(" Generating samples...")
64         # generate_samples(model, n_samples=5) # Assumes this function exists
65
66 # End of epoch - calculate average training loss
67 avg_train_loss = sum(epoch_losses) / len(epoch_losses)
68 train_losses.append(avg_train_loss)
69 print(f"\nTraining - Epoch {epoch+1} average loss: {avg_train_loss:.4f}")
70
71 # Validation phase
72 model.eval()
73 val_epoch_losses = []
74 print("Running validation...")
75
76 with torch.no_grad(): # Disable gradients for validation
77     for val_images, val_labels in val_loader: # Using 'val_loader'
78         val_images = val_images.to(device)
79         val_labels = val_labels.to(device)
80
81         # Calculate validation loss
82         val_loss = train_step(val_images, val_labels) # Pass both images and labels
83         val_epoch_losses.append(val_loss.item())
84
85 # Calculate average validation loss
86 avg_val_loss = sum(val_epoch_losses) / len(val_epoch_losses)
87 val_losses.append(avg_val_loss)
88 print(f"Validation - Epoch {epoch+1} average loss: {avg_val_loss:.4f}")
89
90 # Learning rate scheduling based on validation loss
91 scheduler.step(avg_val_loss)
92 current_lr = optimizer.param_groups[0]['lr']
93 print(f"Learning rate: {current_lr:.6f}")
94
95 # Generate samples at the end of each epoch
96 if epoch % 2 == 0 or epoch == EPOCHS - 1:
97     print("\nGenerating samples for visual progress check...")
98     # generate_samples(model, n_samples=10) # Assumes this function exists
99
100 # Save best model based on validation loss
101 if avg_val_loss < best_loss:
102     best_loss = avg_val_loss
103     # safe_save_model(model, 'best_diffusion_model.pt', optimizer, epoch, best_loss) # Assumes this function exists
104     print(f"\n New best model saved! (Val Loss: {best_loss:.4f})")
105     no_improve_epochs = 0
106 else:
107     no_improve_epochs += 1
108     print(f"No improvement for {no_improve_epochs}/{early_stopping_patience} epochs")
109
110 # Early stopping
111 if no_improve_epochs >= early_stopping_patience:
112     print("\nEarly stopping triggered! No improvement in validation loss.")
113     break
114
115 # Plot loss curves every few epochs
116 if epoch % 5 == 0 or epoch == EPOCHS - 1:
117     plt.figure(figsize=(10, 5))
118     plt.plot(train_losses, label='Training Loss')
119     plt.plot(val_losses, label='Validation Loss')
120     plt.xlabel('Epoch')
121     plt.ylabel('Loss')
122     plt.title('Training and Validation Loss')
123     plt.legend()
124     plt.grid(True)
125     plt.show()
126
127 # Catch errors like user interrupting (Ctrl+C)
128 except KeyboardInterrupt:
129     print("\n" + "="*50)
130     print("TRAINING INTERRUPTED BY USER")
131     print("=".join(["="]*50))
132

```

```
132     print("Saving current model state...")
133     # Use avg_val_loss or last epoch loss for saving
134     last_loss = val_losses[-1] if val_losses else avg_train_loss
135     # safe_save_model(model, 'interrupted_model.pt', optimizer, epoch, last_loss) # Assumes this function exists
136
137 except Exception as e:
138     print("\n" + "="*50)
139     print(f"AN ERROR OCCURRED: {e}")
140     print("="*50)
141     import traceback
142     traceback.print_exc()
143
144 finally:
145     # Final wrap-up
146     print("\n" + "="*50)
147     print("TRAINING COMPLETE")
148     print("="*50)
149     print(f"Best validation loss: {best_loss:.4f}")
150
151     # Generate final samples
152     print("Generating final samples...")
153     # generate_samples(model, n_samples=10) # Assumes this function exists
154
155     # Display final loss curves
156     plt.figure(figsize=(12, 5))
157     plt.plot(train_losses, label='Training Loss')
158     plt.plot(val_losses, label='Validation Loss')
159     plt.xlabel('Epoch')
160     plt.ylabel('Loss')
161     plt.title('Training and Validation Loss')
162     plt.legend()
163     plt.grid(True)
164     plt.show()
165
166     # Clean up memory
167     print("Cleaning up CUDA cache...")
168     torch.cuda.empty_cache()
169     print("Done.")
```

```
1 # (This assumes the following are defined:  
2 # import torch  
3 # import matplotlib.pyplot as plt  
4 # model, device, EPOCHS, train_loader, val_loader  
5 # optimizer, scheduler, train_step  
6 # n_steps, early_stopping_patience, gradient_clip_value,  
7 # display_frequency, generate_frequency  
8 # )  
9 # (It also assumes functions 'generate_samples' and 'safe_save_model' exist,  
10 # but they are commented out below to prevent errors if not defined yet)
```

```

11
12 # Implementation of the main training loop
13 # Training configuration
14 early_stopping_patience = 10 # Number of epochs without improvement before stopping
15 gradient_clip_value = 1.0 # Maximum gradient norm for stability
16 display_frequency = 100 # How often to show progress (in steps)
17 generate_frequency = 500 # How often to generate samples (in steps)
18
19 # Progress tracking variables
20 best_loss = float('inf')
21 train_losses = []
22 val_losses = []
23 no_improve_epochs = 0
24
25 # Training loop
26 print("\n" + "="*50)
27 print("STARTING TRAINING")
28 print("="*50)
29
30 # Wrap the training loop in a try-except block for better error handling
31 try:
32     # This loop starts at the correct (zero) indentation level
33     for epoch in range(EPOCHS):
34         print(f"\nEpoch {epoch+1}/{EPOCHS}")
35         print("-" * 20)
36
37     # Training phase
38     model.train()
39     epoch_losses = []
40
41     # Process each batch
42     for step, (images, labels) in enumerate(train_loader): # Using 'train_loader'
43         images = images.to(device)
44         labels = labels.to(device)
45
46         # Training step
47         optimizer.zero_grad()
48         loss = train_step(images, labels) # Pass both images and labels
49         loss.backward()
50
51         # Add gradient clipping for stability
52         torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=gradient_clip_value)
53
54         optimizer.step()
55         epoch_losses.append(loss.item())
56
57         # Show progress at regular intervals
58         if step % display_frequency == 0:
59             print(f" Step {step}/{len(train_loader)}, Loss: {loss.item():.4f}")
60
61         # Generate samples less frequently to save time
62         if step % generate_frequency == 0 and step > 0:
63             print(" Generating samples...")
64             # generate_samples(model, n_samples=5) # Assumes this function exists
65
66     # End of epoch - calculate average training loss
67     avg_train_loss = sum(epoch_losses) / len(epoch_losses)
68     train_losses.append(avg_train_loss)
69     print(f"\nTraining - Epoch {epoch+1} average loss: {avg_train_loss:.4f}")
70
71     # Validation phase
72     model.eval()
73     val_epoch_losses = []
74     print("Running validation...")
75
76     with torch.no_grad(): # Disable gradients for validation
77         for val_images, val_labels in val_loader: # Using 'val_loader'
78             val_images = val_images.to(device)
79             val_labels = val_labels.to(device)
80
81             # Calculate validation loss
82             val_loss = train_step(val_images, val_labels)
83             val_epoch_losses.append(val_loss.item())
84
85     # Calculate average validation loss
86     avg_val_loss = sum(val_epoch_losses) / len(val_epoch_losses)
87     val_losses.append(avg_val_loss)
88
89

```

```

88     print(f"Validation - Epoch {epoch+1} average loss: {avg_val_loss:.4f}")
89
90     # Learning rate scheduling based on validation loss
91     scheduler.step(avg_val_loss)
92     current_lr = optimizer.param_groups[0]['lr']
93     print(f"Learning rate: {current_lr:.6f}")
94
95     # Generate samples at the end of each epoch
96     if epoch % 2 == 0 or epoch == EPOCHS - 1:
97         print("\nGenerating samples for visual progress check...")
98         # generate_samples(model, n_samples=10) # Assumes this function exists
99
100    # Save best model based on validation loss
101   if avg_val_loss < best_loss:
102       best_loss = avg_val_loss
103       # safe_save_model(model, 'best_diffusion_model.pt', optimizer, epoch, best_loss) # Assumes this func
104       print(f"\n New best model saved! (Val Loss: {best_loss:.4f})")
105       no_improve_epochs = 0
106   else:
107       no_improve_epochs += 1
108   print(f"No improvement for {no_improve_epochs}/{early_stopping_patience} epochs")
109
110  # Early stopping
111  if no_improve_epochs >= early_stopping_patience:
112      print("\nEarly stopping triggered! No improvement in validation loss.")
113      break
114
115  # Plot loss curves every few epochs
116  if epoch % 5 == 0 or epoch == EPOCHS - 1:
117      plt.figure(figsize=(10, 5))
118      plt.plot(train_losses, label='Training Loss')
119      plt.plot(val_losses, label='Validation Loss')
120      plt.xlabel('Epoch')
121      plt.ylabel('Loss')
122      plt.title('Training and Validation Loss')
123      plt.legend()
124      plt.grid(True)
125      plt.show()
126
127 # Catch errors like user interrupting (Ctrl+C)
128 except KeyboardInterrupt:
129     print("\n" + "="*50)
130     print("TRAINING INTERRUPTED BY USER")
131     print("=*50")
132     print("Saving current model state...")
133     # Use avg_val_loss or last epoch loss for saving
134     last_loss = val_losses[-1] if val_losses else avg_train_loss
135     # safe_save_model(model, 'interrupted_model.pt', optimizer, epoch, last_loss) # Assumes this function exists
136
137 except Exception as e:
138     print("\n" + "="*50)
139     print(f"AN ERROR OCCURRED: {e}")
140     print("=*50")
141     import traceback
142     traceback.print_exc()
143
144 finally:
145     # Final wrap-up
146     print("\n" + "="*50)
147     print("TRAINING COMPLETE")
148     print("=*50")
149     print(f"Best validation loss: {best_loss:.4f}")
150
151     # Generate final samples
152     print("Generating final samples...")
153     # generate_samples(model, n_samples=10) # Assumes this function exists
154
155     # Display final loss curves
156     plt.figure(figsize=(12, 5))
157     plt.plot(train_losses, label='Training Loss')
158     plt.plot(val_losses, label='Validation Loss')
159     plt.xlabel('Epoch')
160     plt.ylabel('Loss')
161     plt.title('Training and Validation Loss')
162     plt.legend()
163     plt.grid(True)
164     plt.show()
165

```

```
166 # Clean up memory
167 print("Cleaning up CUDA cache...")
168 torch.cuda.empty_cache()
169 print("Done.")
```

```
1 # (This assumes the following are defined:  
2 # import torch  
3 # import matplotlib.pyplot as plt  
4 # import traceback # Import traceback for error handling  
5 # model, device, EPOCHS, train_loader, val_loader  
6 # optimizer, scheduler, train_step  
7 # n_steps, early_stopping_patience, gradient_clip_value,  
8 # display_frequency, generate_frequency  
9 # )  
10 # (It also assumes functions 'generate_samples' and 'safe_save_model' exist,  
11 # and they are called below)  
12  
13 # Implementation of the main training loop  
14 # Training configuration  
15 early_stopping_patience = 10 # Number of epochs without improvement before stopping  
16 gradient_clip_value = 1.0 # Maximum gradient norm for stability  
17 display_frequency = 100 # How often to show progress (in steps)  
18 generate_frequency = 500 # How often to generate samples (in steps)  
19  
20 # Progress tracking variables  
21 best_loss = float('inf')  
22 train_losses = []  
23 val_losses = []  
24 no_improve_epochs = 0  
25  
26 # Training loop  
27 print("\n" + "="*50)  
28 print("STARTING TRAINING")  
29 print("=". * 50)  
30  
31 # Wrap the training loop in a try-except block for better error handling  
32 try:  
33     # This loop starts at the correct (zero) indentation level  
34     for epoch in range(EPOCHS):  
35         print(f"\nEpoch {epoch+1}/{EPOCHS}")  
36         print("-" * 20)  
37  
38         # Training phase  
39         model.train()  
40         epoch_losses = []  
41  
42         # Process each batch  
43         for step, (images, labels) in enumerate(train_loader): # Using 'train_loader'  
             # ...  
#
```

```

44     images = images.to(device)
45     labels = labels.to(device)
46
47     # Training step
48     optimizer.zero_grad()
49     loss = train_step(images, labels) # Pass both images and labels
50     loss.backward()
51
52     # Add gradient clipping for stability
53     torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=gradient_clip_value)
54
55     optimizer.step()
56     epoch_losses.append(loss.item())
57
58     # Show progress at regular intervals
59     if step % display_frequency == 0:
60         print(f" Step {step}/{len(train_loader)}, Loss: {loss.item():.4f}")
61
62     # Generate samples less frequently to save time
63     if step % generate_frequency == 0 and step > 0:
64         print(" Generating samples...")
65         generate_samples(model, n_samples=5) # Call generate_samples
66
67     # End of epoch - calculate average training loss
68     avg_train_loss = sum(epoch_losses) / len(epoch_losses)
69     train_losses.append(avg_train_loss)
70     print(f"\nTraining - Epoch {epoch+1} average loss: {avg_train_loss:.4f}")
71
72     # Validation phase
73     model.eval()
74     val_epoch_losses = []
75     print("Running validation...")
76
77     with torch.no_grad(): # Disable gradients for validation
78         for val_images, val_labels in val_loader: # Using 'val_loader'
79             val_images = val_images.to(device)
80             val_labels = val_labels.to(device)
81
82             # Calculate validation loss
83             val_loss = train_step(val_images, val_labels) # Pass both images and labels
84             val_epoch_losses.append(val_loss.item())
85
86     # Calculate average validation loss
87     avg_val_loss = sum(val_epoch_losses) / len(val_epoch_losses)
88     val_losses.append(avg_val_loss)
89     print(f"Validation - Epoch {epoch+1} average loss: {avg_val_loss:.4f}")
90
91     # Learning rate scheduling based on validation loss
92     scheduler.step(avg_val_loss)
93     current_lr = optimizer.param_groups[0]['lr']
94     print(f"Learning rate: {current_lr:.6f}")
95
96     # Generate samples at the end of each epoch
97     if epoch % 2 == 0 or epoch == EPOCHS - 1:
98         print("\nGenerating samples for visual progress check...")
99         generate_samples(model, n_samples=10) # Call generate_samples
100
101    # Save best model based on validation loss
102    if avg_val_loss < best_loss:
103        best_loss = avg_val_loss
104        safe_save_model(model, 'best_diffusion_model.pt', optimizer, epoch, best_loss) # Call safe_save_mode
105        print(f"\n New best model saved! (Val Loss: {best_loss:.4f})")
106        no_improve_epochs = 0
107    else:
108        no_improve_epochs += 1
109        print(f"No improvement for {no_improve_epochs}/{early_stopping_patience} epochs")
110
111    # Early stopping
112    if no_improve_epochs >= early_stopping_patience:
113        print("\nEarly stopping triggered! No improvement in validation loss.")
114        break
115
116    # Plot loss curves every few epochs
117    if epoch % 5 == 0 or epoch == EPOCHS - 1:
118        plt.figure(figsize=(10, 5))
119        plt.plot(train_losses, label='Training Loss')
120        plt.plot(val_losses, label='Validation Loss')
121        plt.xlabel('Epoch')

```

```
122         plt.ylabel('Loss')
123         plt.title('Training and Validation Loss')
124         plt.legend()
125         plt.grid(True)
126         plt.show()
127
128 # Catch errors like user interrupting (Ctrl+C)
129 except KeyboardInterrupt:
130     print("\n" + "*50)
131     print("TRAINING INTERRUPTED BY USER")
132     print("*50)
133     print("Saving current model state...")
134     # Use avg_val_loss or last epoch loss for saving
135     last_loss = val_losses[-1] if val_losses else avg_train_loss
136     safe_save_model(model, 'interrupted_model.pt', optimizer, epoch, last_loss) # Call safe_save_model with appr
137
138 except Exception as e:
139     print("\n" + "*50)
140     print(f"AN ERROR OCCURRED: {e}")
141     print("*50)
142     import traceback
143     traceback.print_exc()
144
145 finally:
146     # Final wrap-up
147     print("\n" + "*50)
148     print("TRAINING COMPLETE")
149     print("*50)
150     print(f"Best validation loss: {best_loss:.4f}")
151
152     # Generate final samples
153     print("Generating final samples...")
154     generate_samples(model, n_samples=10) # Call generate_samples
155
156     # Display final loss curves
157     plt.figure(figsize=(12, 5))
158     plt.plot(train_losses, label='Training Loss')
159     plt.plot(val_losses, label='Validation Loss')
160     plt.xlabel('Epoch')
161     plt.ylabel('Loss')
162     plt.title('Training and Validation Loss')
163     plt.legend()
164     plt.grid(True)
165     plt.show()
166
167     # Clean up memory
168     print("Cleaning up CUDA cache...")
169     torch.cuda.empty_cache()
170     print("Done.")
```

Next steps: [Explain error](#)

```
1 import torch  
2 import torch.nn as nn
```

```

3 from einops.layers.torch import Rearrange
4 import torch.nn.functional as F
5
6 # 1. HELPER CLASS: GELUConvBlock
7 class GELUConvBlock(nn.Module):
8     def __init__(self, in_ch, out_ch, group_size):
9         super().__init__()
10        if out_ch % group_size != 0:
11            valid_group_size = group_size
12            while out_ch % valid_group_size != 0 and valid_group_size > 1:
13                valid_group_size -= 1
14            if out_ch % valid_group_size != 0: # Failsafe
15                valid_group_size = 1
16            group_size = valid_group_size
17
18        self.model = nn.Sequential(
19            nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=1),
20            nn.GroupNorm(group_size, out_ch),
21            nn.GELU()
22        )
23    def forward(self, x):
24        return self.model(x)
25
26 # 2. HELPER CLASS: RearrangePoolBlock
27 class RearrangePoolBlock(nn.Module):
28    def __init__(self, in_chs, group_size):
29        super().__init__()
30        # Use named parameters (p1=2, p2=2) to fix the EinopsError
31        self.rearrange = Rearrange('b c (h p1) (w p2) -> b (c p1 p2) h w', p1=2, p2=2)
32        new_chs = in_chs * 4
33
34        if new_chs % group_size != 0:
35            valid_group_size = group_size
36            while new_chs % valid_group_size != 0 and valid_group_size > 1:
37                valid_group_size -= 1
38            if new_chs % valid_group_size != 0: # Failsafe
39                valid_group_size = new_chs
40            group_size = valid_group_size
41
42        self.conv_block = GELUConvBlock(new_chs, new_chs, group_size)
43    def forward(self, x):
44        x = self.rearrange(x)
45        x = self.conv_block(x)
46        return x

```

File /usr/local/cuda/python3.12/lib/python3.12/site-packages/torch/fun/nn/modules/container.py , line 244, in forward

```

1 # (This assumes the following are defined:
2 # import torch
3 # import matplotlib.pyplot as plt
4 # import traceback # Import traceback for error handling
5 # model, device, EPOCHS, train_loader, val_loader
6 # optimizer, scheduler, train_step
7 # n_steps, early_stopping_patience, gradient_clip_value,
8 # display_frequency, generate_frequency
9 #
10 # (It also assumes functions 'generate_samples' and 'safe_save_model' exist,
11 # and they are called below)
12
13 # Implementation of the main training loop
14 # Training configuration
15 early_stopping_patience = 10 # Number of epochs without improvement before stopping
16 gradient_clip_value = 1.0 # Maximum gradient norm for stability
17 display_frequency = 100 # How often to show progress (in steps)
18 generate_frequency = 500 # How often to generate samples (in steps)
19
20 # Progress tracking variables
21 best_loss = float('inf')
22 train_losses = []
23 val_losses = []
24 no_improve_epochs = 0
25
26 # Training loop
27 print("\n" + "*50)
28 print("STARTING TRAINING")
29 print("*50)
30
31 # Wrap the training loop in a try-except block for better error handling
32 try:

```

```

33     # This loop starts at the correct (zero) indentation level
34     for epoch in range(EPOCHS):
35         print(f"\nEpoch {epoch+1}/{EPOCHS}")
36         print("-" * 20)
37
38         # Training phase
39         model.train()
40         epoch_losses = []
41
42         # Process each batch
43         for step, (images, labels) in enumerate(train_loader): # Using 'train_loader'
44             images = images.to(device)
45             labels = labels.to(device)
46
47             # Training step
48             optimizer.zero_grad()
49             loss = train_step(images, labels) # Pass both images and labels
50             loss.backward()
51
52             # Add gradient clipping for stability
53             torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=gradient_clip_value)
54
55             optimizer.step()
56             epoch_losses.append(loss.item())
57
58             # Show progress at regular intervals
59             if step % display_frequency == 0:
60                 print(f" Step {step}/{len(train_loader)}, Loss: {loss.item():.4f}")
61
62             # Generate samples less frequently to save time
63             if step % generate_frequency == 0 and step > 0:
64                 print(" Generating samples...")
65                 generate_samples(model, n_samples=5) # Call generate_samples
66
67             # End of epoch - calculate average training loss
68             avg_train_loss = sum(epoch_losses) / len(epoch_losses)
69             train_losses.append(avg_train_loss)
70             print(f"\nTraining - Epoch {epoch+1} average loss: {avg_train_loss:.4f}")
71
72             # Validation phase
73             model.eval()
74             val_epoch_losses = []
75             print("Running validation...")
76
77             with torch.no_grad(): # Disable gradients for validation
78                 for val_images, val_labels in val_loader: # Using 'val_loader'
79                     val_images = val_images.to(device)
80                     val_labels = val_labels.to(device)
81
82                     # Calculate validation loss
83                     val_loss = train_step(val_images, val_labels) # Pass both images and labels
84                     val_epoch_losses.append(val_loss.item())
85
86             # Calculate average validation loss
87             avg_val_loss = sum(val_epoch_losses) / len(val_epoch_losses)
88             val_losses.append(avg_val_loss)
89             print(f"Validation - Epoch {epoch+1} average loss: {avg_val_loss:.4f}")
90
91             # Learning rate scheduling based on validation loss
92             scheduler.step(avg_val_loss)
93             current_lr = optimizer.param_groups[0]['lr']
94             print(f"Learning rate: {current_lr:.6f}")
95
96             # Generate samples at the end of each epoch
97             if epoch % 2 == 0 or epoch == EPOCHS - 1:
98                 print("\nGenerating samples for visual progress check...")
99                 generate_samples(model, n_samples=10) # Call generate_samples
100
101            # Save best model based on validation loss
102            if avg_val_loss < best_loss:
103                best_loss = avg_val_loss
104                safe_save_model(model, 'best_diffusion_model.pt', optimizer, epoch, best_loss) # Call safe_save_mode
105                print(f"\n New best model saved! (Val Loss: {best_loss:.4f})")
106                no_improve_epochs = 0
107            else:
108                no_improve_epochs += 1
109                print(f"No improvement for {no_improve_epochs}/{early_stopping_patience} epochs")
110
111

```

```

110
111     # Early stopping
112     if no_improve_epochs >= early_stopping_patience:
113         print("\nEarly stopping triggered! No improvement in validation loss.")
114         break
115
116     # Plot loss curves every few epochs
117     if epoch % 5 == 0 or epoch == EPOCHS - 1:
118         plt.figure(figsize=(10, 5))
119         plt.plot(train_losses, label='Training Loss')
120         plt.plot(val_losses, label='Validation Loss')
121         plt.xlabel('Epoch')
122         plt.ylabel('Loss')
123         plt.title('Training and Validation Loss')
124         plt.legend()
125         plt.grid(True)
126         plt.show()
127
128 # Catch errors like user interrupting (Ctrl+C)
129 except KeyboardInterrupt:
130     print("\n" + "="*50)
131     print("TRAINING INTERRUPTED BY USER")
132     print("="*50)
133     print("Saving current model state...")
134     # Use avg_val_loss or last epoch loss for saving
135     last_loss = val_losses[-1] if val_losses else avg_train_loss
136     safe_save_model(model, 'interrupted_model.pt', optimizer, epoch, last_loss) # Call safe_save_model with appr
137
138 except Exception as e:
139     print("\n" + "="*50)
140     print(f"AN ERROR OCCURRED: {e}")
141     print("="*50)
142     import traceback # Make sure traceback is imported
143     traceback.print_exc()
144
145 finally:
146     # Final wrap-up
147     print("\n" + "="*50)
148     print("TRAINING COMPLETE")
149     print("="*50)
150     print(f"Best validation loss: {best_loss:.4f}")
151
152     # Generate final samples
153     print("Generating final samples...")
154     generate_samples(model, n_samples=10) # Call generate_samples
155
156     # Display final loss curves
157     plt.figure(figsize=(12, 5))
158     plt.plot(train_losses, label='Training Loss')
159     plt.plot(val_losses, label='Validation Loss')
160     plt.xlabel('Epoch')
161     plt.ylabel('Loss')
162     plt.title('Training and Validation Loss')
163     plt.legend()
164     plt.grid(True)
165     plt.show()
166
167     # Clean up memory
168     print("Cleaning up CUDA cache...")
169     torch.cuda.empty_cache()
170     print("Done.")

```

```

1 import torch
2 import torch.nn as nn
3 from einops.layers.torch import Rearrange
4 import torch.nn.functional as F
5
6 # 1. HELPER CLASS: GELUConvBlock
7 class GELUConvBlock(nn.Module):
8     def __init__(self, in_ch, out_ch, group_size):
9         super().__init__()
10        # Ensure out_ch is divisible by group_size, adjust if necessary
11        if out_ch % group_size != 0:
12            print(f"Warning: GELUConvBlock out_ch ({out_ch}) not divisible by group_size ({group_size}). Adjusti
13            group_size = min(group_size, out_ch)
14            while out_ch % group_size != 0 and group_size > 1:
15                group_size -= 1

```

```

16         if group_size == 0:
17             group_size = 1 # Prevent division by zero
18             print(f"GELUConvBlock adjusted group_size to {group_size}")
19
20         self.model = nn.Sequential(
21             nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=1),
22             nn.GroupNorm(group_size, out_ch),
23             nn.GELU()
24         )
25     def forward(self, x):
26         return self.model(x)
27
28 # 2. HELPER CLASS: RearrangePoolBlock (kept for completeness, but DownBlock now uses Conv2d)
29 class RearrangePoolBlock(nn.Module):
30     def __init__(self, in_chs, group_size):
31         super().__init__()
32         # Use named parameters (p1=2, p2=2) to fix the EinopsError
33         self.rearrange = Rearrange('b c (h p1) (w p2) -> b (c p1 p2) h w', p1=2, p2=2)
34         new_chs = in_chs * 4
35
36         if new_chs % group_size != 0:
37             valid_group_size = group_size
38             while new_chs % valid_group_size != 0 and valid_group_size > 1:
39                 valid_group_size -= 1
40             if new_chs % valid_group_size != 0: # Failsafe
41                 valid_group_size = new_chs
42             group_size = valid_group_size
43
44         self.conv_block = GELUConvBlock(new_chs, new_chs, group_size)
45     def forward(self, x):
46         x = self.rearrange(x)
47         x = self.conv_block(x)
48         return x
49
50 # 3. HELPER CLASS: DownBlock (Corrected to use Conv2d stride 2 for downsampling)
51 class DownBlock(nn.Module):
52     """
53     Downsampling block for encoding path in U-Net architecture.
54
55     This block:
56     1. Processes input features with two convolutional blocks
57     2. Downsamples spatial dimensions by 2x using a strided convolution.
58
59     Args:
60         in_chs (int): Number of input channels
61         out_chs (int): Number of output channels
62         group_size (int): Number of groups for GroupNorm
63     """
64     def __init__(self, in_chs, out_chs, group_size):
65         super().__init__()
66
67         # Ensure out_chs is divisible by group_size, adjust if necessary
68         if out_chs % group_size != 0:
69             print(f"Warning: DownBlock out_chs ({out_chs}) not divisible by group_size ({group_size}). Adjusting")
70             group_size = min(group_size, out_chs)
71             while out_chs % group_size != 0 and group_size > 1:
72                 group_size -= 1
73             if group_size == 0:
74                 group_size = 1 # Prevent division by zero
75             print(f"DownBlock adjusted group_size to {group_size}")
76
77
78     # Sequential processing of features
79     layers = [
80         # First conv block changes channel dimensions
81         nn.Conv2d(in_chs, out_chs, kernel_size=3, padding=1),
82         nn.GroupNorm(group_size, out_chs),
83         nn.GELU(),
84
85         # Second conv block processes features
86         nn.Conv2d(out_chs, out_chs, kernel_size=3, padding=1),
87         nn.GroupNorm(group_size, out_chs),
88         nn.GELU(),
89
90         # Using Conv2d with stride 2 for robust downsampling instead of RearrangePoolBlock
91         # This layer halves the spatial dimensions (H, W)
92         nn.Conv2d(out_chs, out_chs, kernel_size=4, stride=2, padding=1) # Downsampling
93     ]

```

```

94     self.model = nn.Sequential(*layers)
95
96     # Log the configuration for debugging
97     print(f"Created DownBlock: in_chs={in_chs}, out_chs={out_chs}, spatial_reduction=2x (using Conv2d stride"
98
99     def forward(self, x):
100         """
101             Forward pass through the DownBlock.
102
103             Args:
104                 x (torch.Tensor): Input tensor of shape [B, in_chs, H, W]
105
106             Returns:
107                 torch.Tensor: Output tensor of shape [B, out_chs, H/2, W/2]
108             """
109         return self.model(x)
110
111 # 4. HELPER CLASS: UpBlock
112 class UpBlock(nn.Module):
113     def __init__(self, in_chs, out_chs, group_size):
114         super().__init__()
115         # Ensure out_chs is divisible by group_size, adjust if necessary
116         # Note: The input to the *first* conv block in the sequence is 2 * in_chs
117         if out_chs % group_size != 0:
118             print(f"Warning: UpBlock out_chs ({out_chs}) not divisible by group_size ({group_size}). Adjusting g")
119             group_size_conv = min(group_size, out_chs)
120             while out_chs % group_size_conv != 0 and group_size_conv > 1:
121                 group_size_conv -= 1
122             if group_size_conv == 0:
123                 group_size_conv = 1 # Prevent division by zero
124             print(f"UpBlock adjusted conv group_size to {group_size_conv}")
125         else:
126             group_size_conv = group_size
127
128
129         self.up = nn.ConvTranspose2d(in_chs, in_chs, kernel_size=2, stride=2)
130         self.conv = nn.Sequential(
131             # First block reduces channels from 2*in_chs to out_chs
132             GELUConvBlock(2 * in_chs, out_chs, group_size_conv),
133             # Second block refines the features at the out_chs dimension
134             GELUConvBlock(out_chs, out_chs, group_size_conv)
135         )
136         print(f"Created UpBlock: in_chs={in_chs}, out_chs={out_chs}, spatial_increase=2x")
137
138     def forward(self, x, skip):
139         x_up = self.up(x)
140
141         # Pad x_up if its spatial dimensions are slightly smaller than skip's due to rounding
142         # This can happen with certain image sizes and padding in downsampling
143         if x_up.shape[-2:] != skip.shape[-2:]:
144             # Calculate padding amounts for height and width
145             pad_h = skip.shape[-2] - x_up.shape[-2]
146             pad_w = skip.shape[-1] - x_up.shape[-1]
147             # Apply padding (left, right, top, bottom)
148             x_up = F.pad(x_up, (0, pad_w, 0, pad_h))
149
150
151         x_cat = torch.cat([x_up, skip], dim=1)
152         return self.conv(x_cat)

```

## ▼ 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

```

1 def generate_number(model, number, n_samples=4):
2     """
3         Generate multiple versions of a specific number using the diffusion model.
4
5         Args:

```

```

6     model (nn.Module): The trained diffusion model
7     number (int): The digit to generate (0-9)
8     n_samples (int): Number of variations to generate
9
10    Returns:
11        torch.Tensor: Generated images of shape [n_samples, IMG_CH, IMG_SIZE, IMG_SIZE]
12    """
13    model.eval() # Set model to evaluation mode
14    with torch.no_grad(): # No need for gradients during generation
15        # Start with random noise
16        samples = torch.randn(n_samples, IMG_CH, IMG_SIZE, IMG_SIZE).to(device)
17
18        # Set up the number we want to generate
19        c = torch.full((n_samples,), number, dtype=torch.long).to(device) # Use integer indices (Long)
20
21        # Correctly sized conditioning mask
22        c_mask = torch.ones_like(c.unsqueeze(-1), dtype=torch.float).to(device) # Mask should be Float
23
24
25        # Display progress information
26        print(f"Generating {n_samples} versions of number {number}...")
27
28        # Remove noise step by step
29        for t in range(n_steps-1, -1, -1):
30            t_batch = torch.full((n_samples,), t).to(device)
31            # Pass the class indices 'c' (Long) to remove_noise, not one-hot
32            samples = remove_noise(samples, t_batch, model, c, c_mask) # Fixed: Pass 'c' instead of 'c_one_hot'
33
34            # Optional: Display occasional progress updates
35            if t % (n_steps // 5) == 0:
36                print(f"  Denoising step {n_steps-1-t}/{n_steps-1} completed")
37
38        return samples
39
40 # Generate 4 versions of each number
41 # Make sure you have successfully trained the model in the previous steps first!
42 # plt.figure(figsize=(20, 10))
43 # for i in range(10):
44 #     # Generate samples for current digit
45 #     samples = generate_number(model, i, n_samples=4)
46 #
47 #     # Display each sample
48 #     for j in range(4):
49 #         # Use 2 rows, 10 digits per row, 4 samples per digit
50 #         # i//5 determines the row (0 or 1)
51 #         # i%5 determines the position in the row (0-4)
52 #         # j is the sample index within each digit (0-3)
53 #         plt.subplot(5, 8, (i%5)*8 + (i//5)*4 + j + 1)
54 #
55 #         # Display the image correctly based on channel configuration
56 #         if IMG_CH == 1: # Grayscale
57 #             plt.imshow(samples[j][0].cpu(), cmap='gray')
58 #         else: # Color image
59 #             img = samples[j].permute(1, 2, 0).cpu()
60 #             # Rescale from [-1, 1] to [0, 1] if needed
61 #             if img.min() < 0:
62 #                 img = (img + 1) / 2
63 #             plt.imshow(img)
64 #
65 #         plt.title(f'Digit {i}')
66 #         plt.axis('off')
67 #
68 # plt.tight_layout()
69 # plt.show()
70
71 # STUDENT ACTIVITY: Try generating the same digit with different noise seeds
72 # This shows the variety of styles the model can produce
73 print("\nSTUDENT ACTIVITY: Try generating numbers with different noise seeds after training is complete.")
74
75 # Helper function to generate with seed
76 def generate_with_seed(model, number, seed_value=42, n_samples=10): # Added model argument
77     torch.manual_seed(seed_value)
78     return generate_number(model, number, n_samples)
79
80 # Pick a image and show many variations
81 # Hint select a image e.g. dog # Change this to any other in the dataset of subset you chose
82 # Hint 2 use variations = generate_with_seed

```

```

83 # Hint 3 use plt.figure and plt.imshow to display the variations
84
85 # Example usage (uncomment after model is trained):
86 # digit_to_generate = 7
87 # num_variations = 10
88 # print(f"\nGenerating {num_variations} variations of digit {digit_to_generate} with seed 42:")
89 # variations = generate_with_seed(model, digit_to_generate, seed_value=42, n_samples=num_variations)
90 #
91 # plt.figure(figsize=(num_variations * 2, 2)) # Adjust figure size
92 # for i in range(num_variations):
93 #     plt.subplot(1, num_variations, i+1)
94 #     if IMG_CH == 1:
95 #         plt.imshow(variations[i][0].cpu(), cmap='gray')
96 #     else:
97 #         img = variations[i].permute(1, 2, 0).cpu()
98 #         if img.min() < 0:
99 #             img = (img + 1) / 2
100 #         plt.imshow(img)
101 #     plt.title(f'Var {i+1}')
102 #     plt.axis('off')
103 # plt.tight_layout()
104 # plt.show()

```

STUDENT ACTIVITY: Try generating numbers with different noise seeds after training is complete.

## ▼ 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!

```

1 import torch
2 import torch.nn as nn
3 from einops.layers.torch import Rearrange
4 import torch.nn.functional as F
5
6 # 1. HELPER CLASS: GELUConvBlock (Unchanged)
7 class GELUConvBlock(nn.Module):
8     def __init__(self, in_ch, out_ch, group_size):
9         super().__init__()
10        if out_ch % group_size != 0:
11            valid_group_size = group_size
12            while out_ch % valid_group_size != 0 and valid_group_size > 1:
13                valid_group_size -= 1
14            if out_ch % valid_group_size != 0: valid_group_size = 1
15            group_size = valid_group_size
16        self.model = nn.Sequential(
17            nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=1),
18            nn.GroupNorm(group_size, out_ch),
19            nn.GELU()
20        )
21    def forward(self, x):
22        return self.model(x)
23
24 # 2. HELPER CLASS: RearrangePoolBlock (FIXED)
25 # Now takes 'in_chs' and 'out_chs' and maps in_chs*4 -> out_chs
26 class RearrangePoolBlock(nn.Module):
27    def __init__(self, in_chs, out_chs, group_size):
28        super().__init__()
29        self.rearrange = Rearrange('b c (h p1) (w p2) -> b (c p1 p2) h w', p1=2, p2=2)
30        new_chs = in_chs * 4
31
32        # Fix group_size for new_chs
33        if new_chs % group_size != 0:
34            valid_group_size = group_size
35            while new_chs % valid_group_size != 0 and valid_group_size > 1:
36                valid_group_size -= 1
37            if new_chs % valid_group_size != 0: valid_group_size = new_chs
38            group_size = valid_group_size
39
40        # This conv now correctly maps 4*in_chs -> out_chs
41        self.conv_block = GELUConvBlock(new_chs, out_chs, group_size)
42
43    def forward(self, x):
44        x = self.rearrange(x)
45        x = self.conv_block(x)

```

```

46     return x
47
48 # 3. HELPER CLASS: DownBlock (FIXED)
49 # Now calls the corrected RearrangePoolBlock
50 class DownBlock(nn.Module):
51     def __init__(self, in_chs, out_chs, group_size):
52         super().__init__()
53         layers = [
54             GELUConvBlock(in_chs, out_chs, group_size),
55             GELUConvBlock(out_chs, out_chs, group_size),
56             # This now correctly takes 'out_chs' and outputs 'out_chs'
57             RearrangePoolBlock(out_chs, out_chs, group_size)
58         ]
59         self.model = nn.Sequential(*layers)
60     def forward(self, x):
61         return self.model(x)
62
63 # 4. HELPER CLASS: UpBlock (FIXED)
64 # Now correctly handles different channels from skip connection
65 class UpBlock(nn.Module):
66     # Takes in_chs (from below), skip_chs (from skip), and out_chs
67     def __init__(self, in_chs, skip_chs, out_chs, group_size):
68         super().__init__()
69         self.up = nn.ConvTranspose2d(in_chs, in_chs, kernel_size=2, stride=2)
70         # Conv block now takes (in_chs + skip_chs)
71         self.conv = nn.Sequential(
72             GELUConvBlock(in_chs + skip_chs, out_chs, group_size),
73             GELUConvBlock(out_chs, out_chs, group_size)
74         )
75     def forward(self, x, skip):
76         x_up = self.up(x)
77         x_cat = torch.cat([x_up, skip], dim=1)
78         return self.conv(x_cat)
79
80 # 5. MAIN UNET CLASS (FIXED)
81 # Now calls the corrected UpBlock
82 class UNet(nn.Module):
83     def __init__(self, T, img_ch, img_size, down_chs, t_embed_dim, c_embed_dim):
84         super()

```

## ▼ 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

```

1 ## Step 8: Adding CLIP Evaluation
2
3 # CLIP (Contrastive Language–Image Pre-training) is a powerful model by OpenAI that connects text and images.
4 # We'll use it to evaluate how recognizable our generated digits are by measuring how strongly
5 # the CLIP model associates our generated images with text descriptions like "an image of the digit 7".
6
7 # First, we need to install CLIP and its dependencies
8 print("Setting up CLIP (Contrastive Language–Image Pre-training) model...")
9
10 # Track installation status
11 clip_available = False
12
13 try:
14     # Install dependencies first – these help CLIP process text and images
15     print("Installing CLIP dependencies...")
16     !pip install -q ftfy regex tqdm
17
18     # Install CLIP from GitHub
19     print("Installing CLIP from GitHub repository...")
20     !pip install -q git+https://github.com/openai/CLIP.git
21
22     # Import and verify CLIP is working
23     print("Importing CLIP...")
24     import clip
25
26     # Test that CLIP is functioning
27     model_a = clip.available_models[0]

```