Diffusion Model for Fashion-MNIST

This notebook demonstrates how to train a diffusion model to generate images from the Fashion-MNIST dataset. We implement a U-Net–based architecture conditioned on both time and class labels to gradually remove noise and generate realistic images. Training uses a constant learning rate for steady progress.

Overview:

- **Forward Diffusion:** Gradually add noise to images.
- **Reverse Diffusion:** The model learns to denoise images step by step.
- **Architecture:** A custom U-Net with explicit handling of channel dimensions.
- **Sampling:** Generate clear Fashion-MNIST images from pure noise.

Step 1: Setup and Imports

```
# Install packages if necessary (Kaggle usually preinstalls these)
# !pip install torch torchvision matplotlib tqdm einops

import os
import math
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.optim import Adam
from torch.utils.data import DataLoader, random_split
from torchvision import datasets, transforms
from torchvision.utils import make_grid, save_image
import matplotlib.pyplot as plt
from tqdm.notebook import tqdm
import numpy as np
from einops import rearrange

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

```
Using device: cuda
```

Step 2: Configuration and Diffusion Schedules

```
# ----- Configuration -----
IMG_SIZE = 28          # Fashion-MNIST image size
IMG_CH = 1             # Grayscale images
N_CLASSES = 10         # 10 classes in Fashion-MNIST
BATCH_SIZE = 64
EPOCHS = 30
LEARNING_RATE = 0.001  # Constant learning rate

# Diffusion parameters
TIMESTEPS = 100
BETA_START = 0.0001
BETA_END = 0.02
GRAD_CLIP = 1.0

# Diffusion schedule: linearly spaced betas, then alphas and cumulative product
betas = torch.linspace(BETA_START, BETA_END, TIMESTEPS, device=DEVICE)
alphas = 1.0 - betas
alpha_bars = torch.cumprod(alphas, dim=0)
sqrt_alpha_bars = torch.sqrt(alpha_bars)
sqrt_one_minus_alpha_bars = torch.sqrt(1 - alpha_bars)

print("Configuration set.")
```

```
Configuration set.
```

Step 3: Diffusion Process Functions

```
def add_noise(x0, t):

    noise = torch.randn_like(x0)
    sqrt_alpha_t = sqrt_alpha_bars[t].view(-1, 1, 1, 1)
    sqrt_one_minus_alpha_t = sqrt_one_minus_alpha_bars[t].view(-1, 1, 1, 1)
    x_t = sqrt_alpha_t * x0 + sqrt_one_minus_alpha_t * noise
    return x_t, noise

def remove_noise(x_t, t, model, cond, cond_mask):

    predicted_noise = model(x_t, t, cond, cond_mask)
    alpha_t = alphas[t].view(-1, 1, 1, 1)
    beta_t  = betas[t].view(-1, 1, 1, 1)
    sqrt_one_minus_alpha_t = sqrt_one_minus_alpha_bars[t].view(-1, 1, 1, 1)
    if t.item() == 0:
        return x_t
    mean = (1 / torch.sqrt(alpha_t)) * (x_t - (beta_t / sqrt_one_minus_alpha_t) * predicted_noise)
    noise = torch.randn_like(x_t)
    return mean + torch.sqrt(beta_t) * noise

def sample_image(model, class_label, num_samples=1):

    model.eval()
    with torch.no_grad():
        x = torch.randn(num_samples, IMG_CH, IMG_SIZE, IMG_SIZE, device=DEVICE)
        cond = F.one_hot(torch.tensor([class_label]*num_samples, device=DEVICE), num_classes=N_CLASSES).float()
        cond_mask = torch.ones(num_samples, 1, device=DEVICE)
        for t in reversed(range(TIMESTEPS)):
            t_tensor = torch.full((num_samples,), t, device=DEVICE, dtype=torch.long)
            x = remove_noise(x, t_tensor, model, cond, cond_mask)
        x = (x - x.min()) / (x.max() - x.min() + 1e-8)
        return x
```

Step 4: Model Definition

```
# Custom U-Net for Diffusion with Time and Class Conditioning.
# The UpSample module now accepts the channel count from the skip connection explicitly.

class ConvBlock(nn.Module):
    def __init__(self, in_ch, out_ch, group_size):
        super().__init__()
        while group_size > 1 and out_ch % group_size != 0:
            group_size -= 1
        self.block = nn.Sequential(
            nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=1),
            nn.GroupNorm(num_groups=group_size, num_channels=out_ch),
            nn.GELU()
        )
    def forward(self, x):
        return self.block(x)

class DownSample(nn.Module):
    def __init__(self, in_ch, out_ch, group_size):
        super().__init__()
        self.conv1 = ConvBlock(in_ch, out_ch, group_size)
        self.conv2 = ConvBlock(out_ch, out_ch, group_size)
    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        # Downsample spatially by factor 2; channels multiply by 4.
        x = rearrange(x, 'b c (h r1) (w r2) -> b (c r1 r2) h w', r1=2, r2=2)
        return x

# Modified UpSample: takes skip connection channel count as an argument.
class UpSample(nn.Module):
```

```python
    def __init__(self, in_ch, out_ch, skip_ch, group_size):
        super().__init__()
        self.upconv = nn.ConvTranspose2d(in_ch, out_ch, kernel_size=2, stride=2)
        # After upconv, output shape: [B, out_ch, H*2, W*2]; skip has skip_ch channels.
        self.conv1 = ConvBlock(out_ch + skip_ch, out_ch, group_size)
        self.conv2 = ConvBlock(out_ch, out_ch, group_size)
    def forward(self, x, skip):
        x = self.upconv(x)
        x = torch.cat([x, skip], dim=1)
        x = self.conv1(x)
        x = self.conv2(x)
        return x

# Sinusoidal time embedding, as in Transformers.
class SinusoidalPositionEmbedBlock(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.dim = dim
    def forward(self, t):
        device = t.device
        half_dim = self.dim // 2
        emb = torch.log(torch.tensor(10000.0, device=device)) / (half_dim - 1)
        emb = torch.exp(torch.arange(half_dim, device=device) * -emb)
        emb = t[:, None] * emb[None, :]
        return torch.cat([emb.sin(), emb.cos()], dim=1)

class TimeEmbedding(nn.Module):
    def __init__(self, embed_dim):
        super().__init__()
        self.sinusoid = SinusoidalPositionEmbedBlock(embed_dim)
        self.linear = nn.Sequential(
            nn.Linear(embed_dim, embed_dim),
            nn.GELU()
        )
    def forward(self, t):
        emb = self.sinusoid(t)
        emb = self.linear(emb)
        return emb

class ClassEmbedding(nn.Module):
    def __init__(self, num_classes, embed_dim):
        super().__init__()
        self.embed = nn.Sequential(
            nn.Linear(num_classes, embed_dim),
            nn.GELU(),
            nn.Linear(embed_dim, embed_dim)
        )
    def forward(self, c):
        return self.embed(c)

# DiffusionUNet: Uses two downsampling blocks and two upsampling blocks.
# Skip connections: skip0 from init_conv output, skip1 from first downsample.
class DiffusionUNet(nn.Module):
    def __init__(self, T, img_ch, down_channels, time_embed_dim, num_classes):

        super().__init__()
        self.T = T
        self.time_embed = TimeEmbedding(time_embed_dim)
        self.class_embed = ClassEmbedding(num_classes, time_embed_dim)
        # Initial convolution.
        self.init_conv = ConvBlock(img_ch, down_channels[0], group_size=8)  # Output: [B, 32, 28, 28]

        # Downsampling: We'll do two down samples.
        self.down1 = DownSample(down_channels[0], down_channels[1], group_size=8)
        # After down1: channels become 64*4 = 256, spatial: 14x14.
        self.down2 = DownSample(down_channels[1]*4, down_channels[2], group_size=8)
        # After down2: channels become 128*4 = 512, spatial: 7x7.

        # Bottleneck
        self.bottleneck = nn.Sequential(
            ConvBlock(down_channels[2]*4, down_channels[2]*4, group_size=8),
            ConvBlock(down_channels[2]*4, down_channels[2]*4, group_size=8)
        )
        bottleneck_channels = down_channels[2]*4  # 128*4 = 512.

        # Conditioning projection: map (time_embed + class_embed) from [B, time_embed_dim] to [B, 512]
        self.cond_proj = nn.Sequential(
```

```
            nn.Linear(time_embed_dim, bottleneck_channels),
            nn.GELU()
        )

        # Upsampling: Two up blocks.
        # First up block: Input from bottleneck (512), skip from down2: its input skip is from output of down1, which is 256 channels.
        self.up1 = UpSample(in_ch=bottleneck_channels, out_ch=64, skip_ch=256, group_size=8)
        # Second up block: After up1, output channels: 64. Skip from initial conv: 32 channels.
        self.up2 = UpSample(in_ch=64, out_ch=32, skip_ch=32, group_size=8)

        # Final convolution: from 32 to output channel (1)
        self.final_conv = nn.Conv2d(32, img_ch, kernel_size=1)

    def forward(self, x, t, c, c_mask):
        # Compute conditioning embeddings.
        t_emb = self.time_embed(t)         # [B, time_embed_dim]
        c_emb = self.class_embed(c)          # [B, time_embed_dim]
        cond = t_emb + c_emb                 # [B, time_embed_dim]
        cond = self.cond_proj(cond).unsqueeze(-1).unsqueeze(-1)  # [B, bottleneck_channels, 1, 1]

        # Encoder
        x0 = self.init_conv(x)               # [B, 32, 28, 28] -> skip0
        x1 = self.down1(x0)                  # [B, 256, 14, 14] -> skip1
        x2 = self.down2(x1)                  # [B, 512, 7, 7]

        # Bottleneck
        x2 = self.bottleneck(x2)             # [B, 512, 7, 7]
        # Add conditioning (broadcast across spatial dims)
        x2 = x2 + cond

        # Decoder
        x3 = self.up1(x2, x1)                # up1: from (512 -> 64), uses skip from down1 (256 channels) -> output: [B, 64, 14, 14]
        x4 = self.up2(x3, x0)                # up2: from (64 -> 32), uses skip from init_conv (32 channels) -> output: [B, 32, 28, 28]
        out = self.final_conv(x4)            # Output: [B, img_ch, 28, 28]
        return out

# Alias DiffusionModel to DiffusionUNet.
DiffusionModel = DiffusionUNet
```

Step 5: Data Loading

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

dataset = datasets.FashionMNIST(root='./data', train=True, transform=transform, download=True)
train_size = int(0.8 * len(dataset))
val_size = len(dataset) - train_size
train_dataset, val_dataset = random_split(dataset, [train_size, val_size], generator=torch.Generator().manual_seed(42))

train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=2)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=2)

print(f"Dataset loaded: {len(dataset)} samples (Train: {len(train_dataset)}, Val: {len(val_dataset)})")
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz to ./data/FashionMNIST/raw/train-
100%|██████████| 26.4M/26.4M [00:01<00:00, 15.7MB/s]
Extracting ./data/FashionMNIST/raw/train-images-idx3-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw/train-
100%|██████████| 29.5k/29.5k [00:00<00:00, 267kB/s]
Extracting ./data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to ./data/FashionMNIST/raw/t10k-ir
100%|██████████| 4.42M/4.42M [00:00<00:00, 4.98MB/s]
Extracting ./data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw/t10k-la
```

```
100%|██████████| 5.15k/5.15k [00:00<00:00, 24.7MB/s]Extracting ./data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/FashionM

Dataset loaded: 60000 samples (Train: 48000, Val: 12000)
```

Step 6: Training Loop

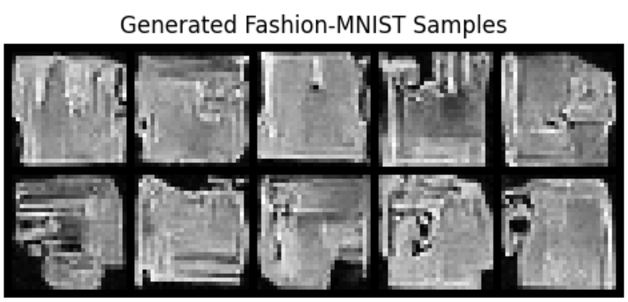
```python
def train_step(model, images, labels):
    # One-hot encode labels.
    c = F.one_hot(labels, num_classes=N_CLASSES).float().to(DEVICE)
    c_mask = torch.ones(labels.size(0), 1, device=DEVICE)
    t = torch.randint(0, TIMESTEPS, (images.size(0),), device=DEVICE, dtype=torch.long)
    x_t, noise = add_noise(images, t)
    pred_noise = model(x_t, t, c, c_mask)
    loss = F.mse_loss(pred_noise, noise)
    return loss

def train_model(model, train_loader, val_loader, optimizer, epochs):
    best_val_loss = float('inf')
    for epoch in range(epochs):
        model.train()
        train_loss = 0.0
        for images, labels in tqdm(train_loader, desc=f"Epoch {epoch+1}/{epochs} [Training]"):
            images = images.to(DEVICE)
            labels = labels.to(DEVICE)
            optimizer.zero_grad()
            loss = train_step(model, images, labels)
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), GRAD_CLIP)
            optimizer.step()
            train_loss += loss.item()
        avg_train_loss = train_loss / len(train_loader)

        model.eval()
        val_loss = 0.0
        with torch.no_grad():
            for images, labels in tqdm(val_loader, desc=f"Epoch {epoch+1}/{epochs} [Validation]"):
                images = images.to(DEVICE)
                labels = labels.to(DEVICE)
                loss = train_step(model, images, labels)
                val_loss += loss.item()
        avg_val_loss = val_loss / len(val_loader)
        print(f"Epoch {epoch+1}/{epochs} - Train Loss: {avg_train_loss:.4f}, Val Loss: {avg_val_loss:.4f}")

        if avg_val_loss < best_val_loss:
            best_val_loss = avg_val_loss
            checkpoint = {
                'epoch': epoch + 1,
                'model_state_dict': model.state_dict(),
                'optimizer_state_dict': optimizer.state_dict(),
                'train_loss': avg_train_loss,
                'val_loss': avg_val_loss
            }
            torch.save(checkpoint, f"checkpoint_epoch_{epoch+1}.pt")
            print(f"Checkpoint saved at epoch {epoch+1}")

model = DiffusionModel(
    T=TIMESTEPS,
    img_ch=IMG_CH,
    down_channels=(32, 64, 128),
    time_embed_dim=8,
    num_classes=N_CLASSES
).to(DEVICE)
optimizer = Adam(model.parameters(), lr=LEARNING_RATE, weight_decay=1e-5)
print("Starting training (constant learning rate)...")
train_model(model, train_loader, val_loader, optimizer, EPOCHS)
```

```
Starting training (constant learning rate)...
Epoch 1/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 1/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 1/30 - Train Loss: 0.1695, Val Loss: 0.1400
Checkpoint saved at epoch 1
Epoch 2/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 2/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 2/30 - Train Loss: 0.1313, Val Loss: 0.1314
Checkpoint saved at epoch 2
Epoch 3/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 3/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 3/30 - Train Loss: 0.1236, Val Loss: 0.1212
Checkpoint saved at epoch 3
Epoch 4/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 4/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 4/30 - Train Loss: 0.1209, Val Loss: 0.1188
Checkpoint saved at epoch 4
Epoch 5/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 5/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 5/30 - Train Loss: 0.1181, Val Loss: 0.1188
Epoch 6/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 6/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 6/30 - Train Loss: 0.1165, Val Loss: 0.1154
Checkpoint saved at epoch 6
Epoch 7/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 7/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 7/30 - Train Loss: 0.1156, Val Loss: 0.1135
Checkpoint saved at epoch 7
Epoch 8/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 8/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 8/30 - Train Loss: 0.1133, Val Loss: 0.1156
Epoch 9/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 9/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 9/30 - Train Loss: 0.1128, Val Loss: 0.1117
Checkpoint saved at epoch 9
Epoch 10/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 10/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 10/30 - Train Loss: 0.1130, Val Loss: 0.1127
Epoch 11/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 11/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 11/30 - Train Loss: 0.1120, Val Loss: 0.1118
Epoch 12/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 12/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 12/30 - Train Loss: 0.1114, Val Loss: 0.1121
Epoch 13/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 13/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 13/30 - Train Loss: 0.1112, Val Loss: 0.1102
Checkpoint saved at epoch 13
Epoch 14/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 14/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 14/30 - Train Loss: 0.1108, Val Loss: 0.1128
Epoch 15/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 15/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 15/30 - Train Loss: 0.1108, Val Loss: 0.1107
Epoch 16/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 16/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 16/30 - Train Loss: 0.1102, Val Loss: 0.1116
Epoch 17/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 17/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 17/30 - Train Loss: 0.1103, Val Loss: 0.1105
Epoch 18/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 18/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 18/30 - Train Loss: 0.1095, Val Loss: 0.1108
Epoch 19/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 19/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 19/30 - Train Loss: 0.1091, Val Loss: 0.1075
Checkpoint saved at epoch 19
Epoch 20/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 20/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 20/30 - Train Loss: 0.1099, Val Loss: 0.1110
Epoch 21/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 21/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 21/30 - Train Loss: 0.1094, Val Loss: 0.1091
Epoch 22/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 22/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 22/30 - Train Loss: 0.1077, Val Loss: 0.1104
Epoch 23/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 23/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 23/30 - Train Loss: 0.1098, Val Loss: 0.1093
Epoch 24/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 24/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 24/30 - Train Loss: 0.1084, Val Loss: 0.1095
Epoch 25/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 25/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
```

```
Epoch 25/30 - Train Loss: 0.1089, Val Loss: 0.1092
Epoch 26/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 26/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 26/30 - Train Loss: 0.1092, Val Loss: 0.1098
Epoch 27/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 27/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 27/30 - Train Loss: 0.1081, Val Loss: 0.1097
Epoch 28/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 28/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 28/30 - Train Loss: 0.1083, Val Loss: 0.1082
Epoch 29/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 29/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 29/30 - Train Loss: 0.1084, Val Loss: 0.1069
Checkpoint saved at epoch 29
Epoch 30/30 [Training]:   0%|          | 0/750 [00:00<?, ?it/s]
Epoch 30/30 [Validation]:   0%|          | 0/188 [00:00<?, ?it/s]
Epoch 30/30 - Train Loss: 0.1075, Val Loss: 0.1109
```

Step 7: Sampling and Generating Images

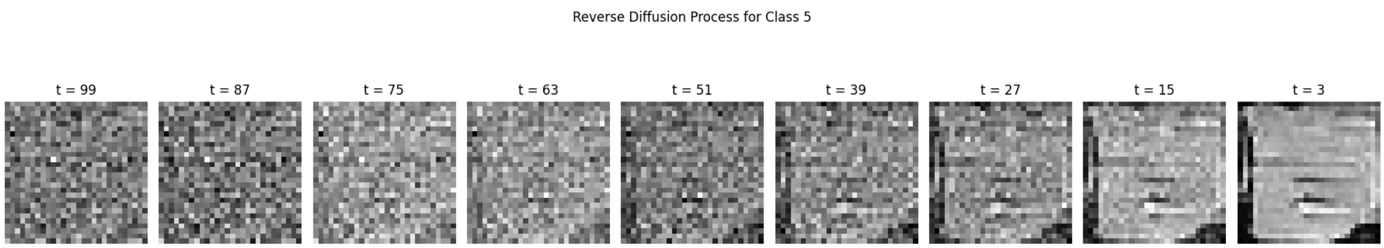
```
def generate_samples(model, num_samples_per_class=1):
    model.eval()
    samples = []
    with torch.no_grad():
        for class_label in range(N_CLASSES):
            x = sample_image(model, class_label, num_samples=num_samples_per_class)
            samples.append(x)
        samples = torch.cat(samples, dim=0)
        grid = make_grid(samples, nrow=5, normalize=True)
        plt.figure(figsize=(6,6))
        if IMG_CH == 1:
            plt.imshow(grid.permute(1,2,0).cpu().squeeze(), cmap='gray')
        else:
            plt.imshow(grid.permute(1,2,0).cpu())
        plt.axis('off')
        plt.title("Generated Fashion-MNIST Samples")
        plt.show()

generate_samples(model)
```



Step 8: Visualizing the Diffusion Process

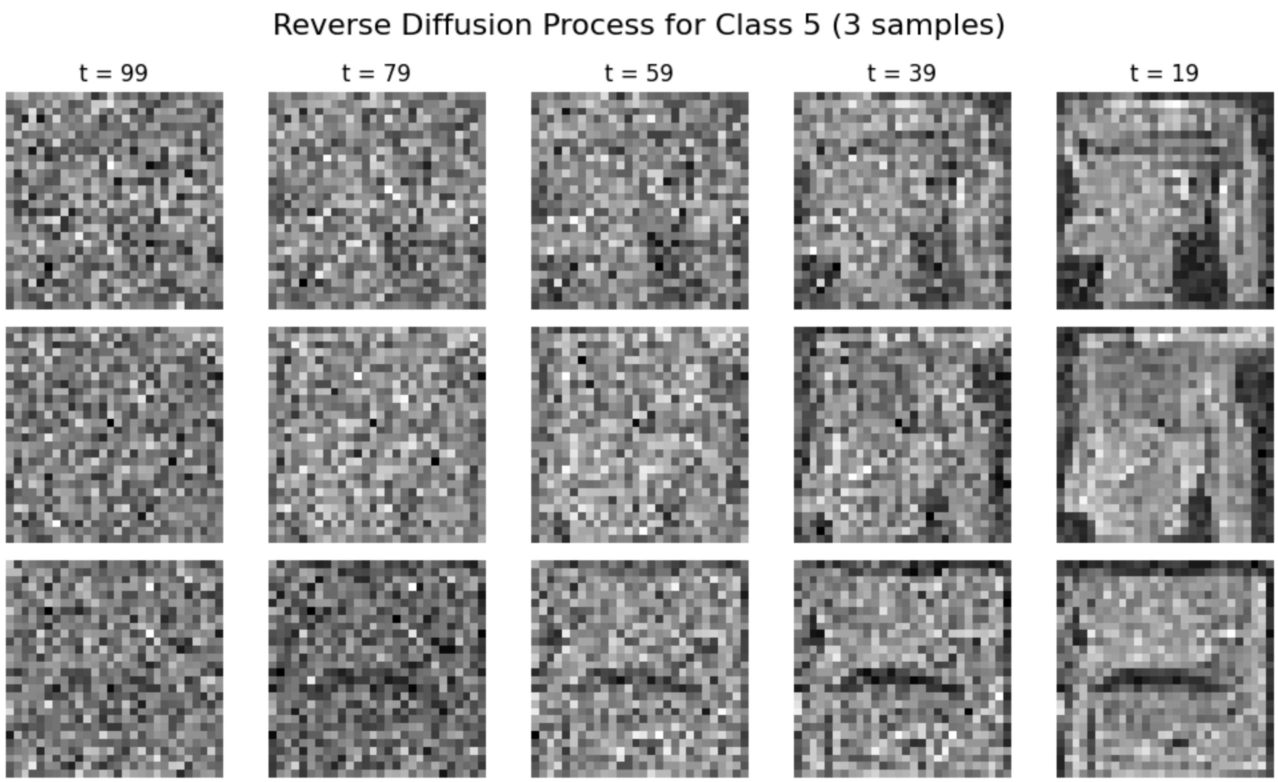
```
def visualize_diffusion(model, class_label=5, steps=8):
    model.eval()
    with torch.no_grad():
        x = torch.randn(1, IMG_CH, IMG_SIZE, IMG_SIZE, device=DEVICE)
        cond = F.one_hot(torch.tensor([class_label], device=DEVICE), num_classes=N_CLASSES).float()
        cond_mask = torch.ones(1, 1, device=DEVICE)
        vis_steps = list(range(TIMESTEPS-1, -1, -max(1, TIMESTEPS//steps)))
        vis_steps = sorted(vis_steps, reverse=True)
        images = []
        for t in range(TIMESTEPS-1, -1, -1):
            t_tensor = torch.full((1,), t, device=DEVICE, dtype=torch.long)
            x = remove_noise(x, t_tensor, model, cond, cond_mask)
            if t in vis_steps:
                norm_x = (x - x.min()) / (x.max() - x.min() + 1e-8)
                images.append(norm_x.cpu())
        num_imgs = len(images)
```

```
        plt.figure(figsize=(num_imgs * 2, 4))
        for i, img in enumerate(images):
            plt.subplot(1, num_imgs, i+1)
            if IMG_CH == 1:
                plt.imshow(img.squeeze(), cmap='gray')
            else:
                plt.imshow(img.permute(1,2,0))
            plt.axis('off')
            plt.title(f"t = {vis_steps[i]}")
        plt.suptitle(f"Reverse Diffusion Process for Class {class_label}")
        plt.tight_layout()
        plt.show()

visualize_diffusion(model, class_label=5, steps=8)
```




```
def visualize_diffusion_multisample(model, class_label=5, num_samples=3, steps=5):
    """
    Visualize the reverse diffusion process for multiple samples of a given class.

    For each sample, this function collects images at `steps` evenly spaced timesteps
    during the reverse diffusion process and then displays them in a grid. Each row
    corresponds to a different sample and each column to a particular timestep.

    Args:
        model: The trained diffusion model.
        class_label: The Fashion-MNIST class to generate (0-9).
        num_samples: Number of samples to generate for the chosen class.
        steps: Number of diffusion steps (timesteps) to capture per sample.
    """
    model.eval()
    all_visuals = []  # List to store images for each sample
    # Determine the timesteps to capture. E.g., if steps=5, get 5 evenly spaced timesteps.
    vis_timesteps = list(range(TIMESTEPS-1, -1, -max(1, TIMESTEPS // steps)))
    vis_timesteps = sorted(vis_timesteps, reverse=True)

    for sample_idx in range(num_samples):
        with torch.no_grad():
            # Start with a new random noise sample.
            x = torch.randn(1, IMG_CH, IMG_SIZE, IMG_SIZE, device=DEVICE)
            # Create one-hot conditioning for the chosen class.
            cond = F.one_hot(torch.tensor([class_label], device=DEVICE), num_classes=N_CLASSES).float()
            cond_mask = torch.ones(1, 1, device=DEVICE)
            sample_images = []  # Images for current sample.
            # Run reverse diffusion, capturing images at specified timesteps.
            for t in range(TIMESTEPS-1, -1, -1):
                t_tensor = torch.full((1,), t, device=DEVICE, dtype=torch.long)
                x = remove_noise(x, t_tensor, model, cond, cond_mask)
                if t in vis_timesteps:
                    norm_x = (x - x.min()) / (x.max() - x.min() + 1e-8)
                    sample_images.append(norm_x.cpu())
        all_visuals.append(sample_images)

    # Create a grid: rows correspond to samples; columns correspond to the chosen timesteps.
    num_timesteps = len(vis_timesteps)
    fig, axes = plt.subplots(nrows=num_samples, ncols=num_timesteps, figsize=(num_timesteps * 2, num_samples * 2))

    for i in range(num_samples):
```

```
        for j in range(num_timesteps):
            # Each sample image is a tensor of shape [IMG_CH, H, W]. For grayscale, squeeze extra channel.
            img = all_visuals[i][j][0]  # Taking the first (and only) channel if necessary.
            ax = axes[i, j] if num_samples > 1 else axes[j]
            if IMG_CH == 1:
                ax.imshow(img.squeeze(), cmap='gray')
            else:
                ax.imshow(img.permute(1, 2, 0))
            ax.axis("off")
            if i == 0:
                ax.set_title(f"t = {vis_timesteps[j]}")
    plt.suptitle(f"Reverse Diffusion Process for Class {class_label} ({num_samples} samples)", fontsize=16)
    plt.tight_layout()
    plt.show()

# To visualize for a given class (e.g. class 5) with 3 samples and 5 timesteps per sample:
visualize_diffusion_multisample(model, class_label=5, num_samples=3, steps=5)
```

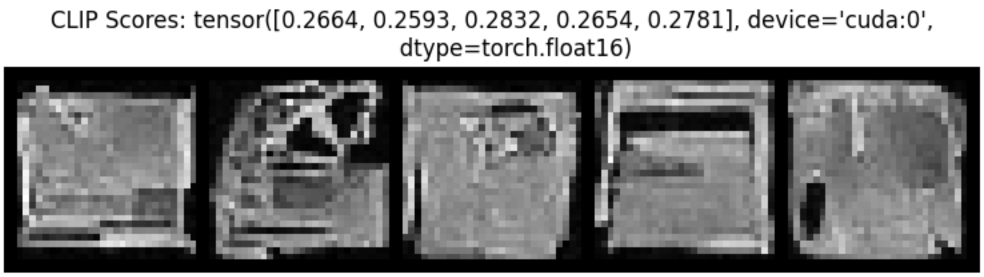



```
# ---- Step 9: CLIP Evaluation (Optional Bonus) ----
# Install CLIP dependencies (uncomment if needed)
!pip install ftfy regex tqdm
!pip install git+https://github.com/openai/CLIP.git


import clip
import torchvision.transforms as transforms

# Load the CLIP model (using the ViT-B/32 variant)
clip_model, clip_preprocess = clip.load("ViT-B/32", device=DEVICE)
clip_model.eval()
print("CLIP model loaded.")

# --- Fix the reverse diffusion functions for batch processing ---
def remove_noise_fixed(x_t, t, model, cond, cond_mask):
    """
    Revised reverse diffusion function with batch-friendly check.
    Instead of t.item(), we check t[0].item() so that when t is a vector, it works correctly.
    """
    predicted_noise = model(x_t, t, cond, cond_mask)
    alpha_t = alphas[t].view(-1, 1, 1, 1)
    beta_t  = betas[t].view(-1, 1, 1, 1)
    sqrt_one_minus_alpha_t = sqrt_one_minus_alpha_bars[t].view(-1, 1, 1, 1)
    if t[0].item() == 0:
```

```
        return x_t
    mean = (1 / torch.sqrt(alpha_t)) * (x_t - (beta_t / sqrt_one_minus_alpha_t) * predicted_noise)
    noise = torch.randn_like(x_t)
    return mean + torch.sqrt(beta_t) * noise

def sample_image_fixed(model, class_label, num_samples=1):
    """
    Generates sample images for a given class label by running the reverse diffusion process
    using the fixed version of remove_noise. Returns normalized images in the [0,1] range.
    """
    model.eval()
    with torch.no_grad():
        x = torch.randn(num_samples, IMG_CH, IMG_SIZE, IMG_SIZE, device=DEVICE)
        cond = F.one_hot(torch.tensor([class_label] * num_samples, device=DEVICE), num_classes=N_CLASSES).float()
        cond_mask = torch.ones(num_samples, 1, device=DEVICE)
        # Use the fixed remove_noise function
        for t in reversed(range(TIMESTEPS)):
            t_tensor = torch.full((num_samples,), t, device=DEVICE, dtype=torch.long)
            x = remove_noise_fixed(x, t_tensor, model, cond, cond_mask)
        # Normalize the output for display.
        x = (x - x.min()) / (x.max() - x.min() + 1e-8)
        return x

# Define CLIP evaluation functions.
def evaluate_generated_images_clip(clip_model, samples, prompt):
    """
    Evaluate generated images using CLIP by computing cosine similarity between
    the image embeddings and the embedding of the given text prompt.

    Args:
        clip_model: The CLIP model.
        samples: Tensor of generated images, shape [B, C, H, W], expected in [0,1].
        prompt: A string prompt (e.g., "a clear image of a Fashion-MNIST class 5")

    Returns:
        similarity: Tensor of similarity scores for each image.
    """
    # Resize images to 224x224 (CLIP's input resolution).
    samples_resized = torch.nn.functional.interpolate(samples, size=(224, 224), mode='bilinear', align_corners=False)
    # Convert grayscale images (1 channel) to 3-channel RGB.
    if samples_resized.shape[1] == 1:
        samples_rgb = samples_resized.repeat(1, 3, 1, 1)
    else:
        samples_rgb = samples_resized

    # Normalize using CLIP's standard mean and std.
    normalize = transforms.Normalize((0.48145466, 0.4578275, 0.40821073),
                                     (0.26862954, 0.26130258, 0.27577711))
    # Apply normalization on each image.
    samples_norm = torch.stack([normalize(img) for img in samples_rgb])

    with torch.no_grad():
        image_features = clip_model.encode_image(samples_norm)
        text_tokens = clip.tokenize([prompt]).to(DEVICE)
        text_features = clip_model.encode_text(text_tokens)

    # Normalize features.
    image_features = image_features / image_features.norm(dim=-1, keepdim=True)
    text_features = text_features / text_features.norm(dim=-1, keepdim=True)
    similarity = (image_features @ text_features.T).squeeze(1)
    return similarity

def clip_evaluation_example(model, class_label, num_samples=5):
    """
    Generate a batch of samples for a specific class and evaluate them using CLIP.
    Displays the CLIP similarity scores and the grid of images.

    Args:
        model: The trained diffusion model.
        class_label: Integer (0-9) indicating which Fashion-MNIST class to generate.
        num_samples: Number of generated samples.
    """
    samples = sample_image_fixed(model, class_label, num_samples=num_samples)
    prompt = f"a clear, well-detailed Fashion-MNIST image of class {class_label}"
    scores = evaluate_generated_images_clip(clip_model, samples, prompt)
    print(f"CLIP Similarity Scores for class {class_label}:\n{scores}")
```

```
    # Display images in a grid.
    grid = make_grid(samples, nrow=num_samples, normalize=True)
    plt.figure(figsize=(num_samples * 2, 2))
    if IMG_CH == 1:
        plt.imshow(grid.permute(1, 2, 0).cpu().squeeze(), cmap='gray')
    else:
        plt.imshow(grid.permute(1, 2, 0).cpu())
    plt.title(f"CLIP Scores: {scores}")
    plt.axis("off")
    plt.show()

# Example usage: Evaluate generated images for Fashion-MNIST class 5.
clip_evaluation_example(model, class_label=5, num_samples=5)
```

```
CLIP model loaded.
CLIP Similarity Scores for class 5:
tensor([0.2664, 0.2593, 0.2832, 0.2654, 0.2781], device='cuda:0',
       dtype=torch.float16)
```



Final Remarks and Analysis

Training Analysis:

- The model is trained with a **constant learning rate** for consistent progress.
- The U-Net is conditioned on **time** and **class labels**, with fixed channel dimensions in skip connections.
- Generated samples and diffusion process visualization will help verify that the reverse diffusion process gradually denoises the images.

If everything is working as expected, you should see generated Fashion-MNIST images that clearly resemble the dataset.

Happy training and generating!