Jade Sanchez

ITAI 2376

4/9/2025

Notebook

Dataset Setup and Preparation

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

# Data transforms
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Lambda(lambda x: (x - 0.5) * 2)  # normalize to [-1, 1]
])

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

# Split into train and validation
train_size = int(0.9 * len(dataset))
val_size = len(dataset) - train_size
train_dataset, val_dataset = random_split(dataset, [train_size, val_size])

# Create data loaders
batch_size = 128
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size)
```

Model Architecture – U-Net Skeleton

```
import torch.nn as nn

class GELUConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.block = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
            nn.GELU(),
            nn.BatchNorm2d(out_channels),
        )

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

Basic Down/Up blocks

```
class DownBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.block = nn.Sequential(
            GELUConvBlock(in_channels, out_channels),
            nn.MaxPool2d(2)
        )

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

class UpBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.upsample = nn.ConvTranspose2d(in_channels, out_channels, 2, stride=2)
        self.conv = GELUConvBlock(in_channels, out_channels)

    def forward(self, x, skip_connection):
        x = self.upsample(x)
        x = torch.cat((x, skip_connection), dim=1)
        return self.conv(x)
```

Now the full U-Net

```
class UNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.down1 = DownBlock(1, 64)
        self.down2 = DownBlock(64, 128)
        self.bottom = GELUConvBlock(128, 256)
        self.up2 = UpBlock(256, 128)
        self.up1 = UpBlock(128, 64)
        self.out = nn.Conv2d(64, 1, kernel_size=1)

    def forward(self, x):
        d1 = self.down1(x)
        d2 = self.down2(d1)
        b = self.bottom(d2)
        u2 = self.up2(b, d2)
        u1 = self.up1(u2, d1)
        return self.out(u1)
```

Diffusion Process

```
def forward_diffusion_sample(x_0, t, noise=None):
    if noise is None:
        noise = torch.randn_like(x_0)
    sqrt_alpha_hat = torch.sqrt(alphas_hat[t])[:, None, None, None]
    sqrt_one_minus_alpha_hat = torch.sqrt(1 - alphas_hat[t])[:, None, None, None]
    return sqrt_alpha_hat * x_0 + sqrt_one_minus_alpha_hat * noise, noise
```

noise schedule

```
T = 300  # number of diffusion steps
betas = torch.linspace(1e-4, 0.02, T)
alphas = 1 - betas
alphas_hat = torch.cumprod(alphas, dim=0)
```

Training Loop

```
import torch.nn.functional as F
```

```
model = UNet().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

for epoch in range(num_epochs):
    for images, _ in train_loader:
        images = images.to(device)
        t = torch.randint(0, T, (images.size(0),), device=device).long()
        x_t, noise = forward_diffusion_sample(images, t)
        predicted_noise = model(x_t)
        loss = F.mse_loss(predicted_noise, noise)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    print(f"Epoch {epoch+1} - Loss: {loss.item():.4f}")
```

CLIP Evaluation

```
!pip install git+https://github.com/openai/CLIP.git
import clip
import torchvision.transforms as transforms
from PIL import Image
import torch

device = "cuda" if torch.cuda.is_available() else "cpu"
model_clip, preprocess = clip.load("ViT-B/32", device=device)
```

define a function to evaluate generated images using CLIP

```
def evaluate_with_clip(images, class_labels):
    # images: list of PIL Images
    # class_labels: list of strings (e.g., ["zero", "one", ...])
    text_inputs = clip.tokenize(class_labels).to(device)
    image_inputs = torch.stack([preprocess(img).to(device) for img in images])

    with torch.no_grad():
        image_features = model_clip.encode_image(image_inputs)
        text_features = model_clip.encode_text(text_inputs)

        image_features /= image_features.norm(dim=-1, keepdim=True)
        text_features /= text_features.norm(dim=-1, keepdim=True)

        similarity = (100.0 * image_features @ text_features.T)

    return similarity.softmax(dim=-1).cpu().numpy()  # Confidence scores per label
```

Example usage

```
# If you have a list of PIL images from your generator:
labels = ["zero", "one", "two", "three", "four", "five", "six", "seven", "eight", "nine"]
clip_scores = evaluate_with_clip(generated_images, labels)

# Show confidence per image
for i, score in enumerate(clip_scores):
    print(f"Image {i} - Top Label: {labels[score.argmax()]} (Confidence:
{score.max()*100:.2f}%)")
```