Title

Evaluating Generative Adversarial Networks on MedMNIST Using Quantitative Metrics and TensorBoard

Objective

- To generate synthetic medical images using a GAN trained on the MedMNIST dataset.
- To evaluate the GAN performance with **Inception Score (IS)** and **Fréchet Inception Distance (FID)**.
- To visualize the generated images and evaluation metrics with **TensorBoard**.

Introduction

Generative Adversarial Networks (GANs) have shown significant potential in augmenting medical imaging data. This project evaluates GAN performance by employing:

- **Inception Score (IS):** Measures both image quality and diversity.
- **Fréchet Inception Distance (FID):** Assesses the statistical similarity between generated and real images.

Methodology

1. **Data Generation:**
   The GAN generator produces synthetic MedMNIST images from random latent vectors.
2. **Preprocessing:**
   Images are normalized and formatted (expanded from grayscale to 3 channels) for metric compatibility.
3. **Evaluation:**
   - IS and FID are computed using TorchMetrics.
   - Results are logged and visualized using TensorBoard.

```
import os
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
import torchvision.utils as vutils
from torch.utils.data import DataLoader
from torch.utils.tensorboard import SummaryWriter
from medmnist import PathMNIST
import numpy as np
from scipy.linalg import sqrtm
from torchmetrics.image.inception import InceptionScore
from torchmetrics.image.fid import FrechetInceptionDistance
import torch_fidelity
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
from medmnist import PathMNIST
```

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

```
dataset = PathMNIST(root=r"C:\Users\Simrann\Downloads\qwerty\gans\assignment4\root", split="train", transform=transform)
```

```
from torch.utils.data import Subset
subset_indices = np.random.choice(len(dataset), 10000, replace=False)
subset_dataset = Subset(dataset, subset_indices)
dataloader = DataLoader(subset_dataset, batch_size=64, shuffle=True)
dataset
```

```
Dataset PathMNIST of size 28 (pathmnist)
    Number of datapoints: 89996
    Root location: C:\Users\Simrann\Downloads\qwerty\gans\assignment4\root
    Split: train
    Task: multi-class
    Number of channels: 3
```

```
    Meaning of labels: {'0': 'adipose', '1': 'background', '2': 'debris', '3': 'lymphocytes', '4': 'mucus', '5': 'smooth muscle', '6':
'normal colon mucosa', '7': 'cancer-associated stroma', '8': 'colorectal adenocarcinoma epithelium'}
    Number of samples: {'train': 89996, 'val': 10004, 'test': 7180}
    Description: The PathMNIST is based on a prior study for predicting survival from colorectal cancer histology slides, providing a
dataset (NCT-CRC-HE-100K) of 100,000 non-overlapping image patches from hematoxylin & eosin stained histological images, and a test
dataset (CRC-VAL-HE-7K) of 7,180 image patches from a different clinical center. The dataset is comprised of 9 types of tissues,
resulting in a multi-class classification task. We resize the source images of 3×224×224 into 3×28×28, and split NCT-CRC-HE-100K into
training and validation set with a ratio of 9:1. The CRC-VAL-HE-7K is treated as the test set.
    License: CC BY 4.0
```

```python
for img, _ in dataloader:
    print(f"Image shape: {img.shape}")
    break
```

```
Image shape: torch.Size([64, 3, 28, 28])
```

```python
class Generator(nn.Module):
    def __init__(self, latent_dim):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(latent_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, 28 * 28),
            nn.Tanh()
        )

    def forward(self, z):
        img = self.model(z)
        img = img.view(img.size(0), 1, 28, 28)  # Original shape: (batch_size, 784)
        img = img.repeat(1, 3, 1, 1)          # Repeat the single channel 3 times
        return img
```

```python
#Discriminator for LS-GAN
# Fix for RGB images in MedMNIST (28x28x3 = 2352)
class DiscriminatorLS(nn.Module):
    def __init__(self):
        super(DiscriminatorLS, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(28 * 28 * 3, 512),  # Change 784 → 28*28*3 = 2352
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 1)
        )

    def forward(self, img):
        return self.model(img.view(img.size(0), -1))  # Flatten properly
```

```python
# Define WGAN and WGAN-GP Discriminator
class DiscriminatorW(nn.Module):
    def __init__(self):
        super(DiscriminatorW, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(28 * 28 * 3, 512),  # Change 784 → 28*28*3 = 2352
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 1)
        )

    def forward(self, img):
        return self.model(img.view(img.size(0), -1))  # Flatten properly
```

```python
# Training function
def train_gan(generator, discriminator, g_optimizer, d_optimizer, loss_fn, epochs, gan_type):
    writer = SummaryWriter(log_dir=f'logs/{gan_type}')
    for epoch in range(epochs):
        for i, (real_imgs, _) in enumerate(dataloader):
            real_imgs = real_imgs.to(device)
```

```
            # Train Discriminator
            z = torch.randn(real_imgs.size(0), 100).to(device)
            fake_imgs = generator(z).detach()
            d_real = discriminator(real_imgs)
            d_fake = discriminator(fake_imgs)

            d_loss = 0

            if gan_type == "LS-GAN":
                d_loss = 0.5 * ((d_real - 1) ** 2).mean() + 0.5 * (d_fake ** 2).mean()
            elif gan_type == "WGAN":
                d_loss = -torch.mean(d_real) + torch.mean(d_fake)
            elif gan_type == "WGAN-GP":
                lambda_gp = 10
                epsilon = torch.rand(real_imgs.size(0), 1, 1, 1).to(device)
                x_hat = (epsilon * real_imgs + (1 - epsilon) * fake_imgs).requires_grad_(True)
                d_x_hat = discriminator(x_hat)
                gradients = torch.autograd.grad(outputs=d_x_hat, inputs=x_hat,
                                                grad_outputs=torch.ones_like(d_x_hat),
                                                create_graph=True, retain_graph=True)[0]
                gp = ((gradients.norm(2, dim=1) - 1) ** 2).mean()
                d_loss += lambda_gp * gp

            d_optimizer.zero_grad()
            d_loss.backward()
            d_optimizer.step()

            # Train Generator
            z = torch.randn(real_imgs.size(0), 100).to(device)
            fake_imgs = generator(z)
            g_fake = discriminator(fake_imgs)

            if gan_type == "LS-GAN":
                g_loss = 0.5 * ((g_fake - 1) ** 2).mean()
            else:
                g_loss = -torch.mean(g_fake)

            g_optimizer.zero_grad()
            g_loss.backward()
            g_optimizer.step()

        # Save progress
        if epoch % 10 == 0:
            vutils.save_image(fake_imgs, f"generated_{gan_type}_{epoch}.png", normalize=True)
            writer.add_scalar(f'{gan_type}/D_Loss', d_loss.item(), epoch)
            writer.add_scalar(f'{gan_type}/G_Loss', g_loss.item(), epoch)

    writer.close()
```

```
ls_generator = Generator(100).to(device)
ls_discriminator = DiscriminatorLS().to(device)
g_optimizer_ls = optim.Adam(ls_generator.parameters(), lr=0.0002, betas=(0.5, 0.999))
d_optimizer_ls = optim.Adam(ls_discriminator.parameters(), lr=0.0002, betas=(0.5, 0.999))
```

```
generator_w = Generator(100).to(device)
discriminator_w = DiscriminatorW().to(device)
g_optimizer_w = optim.RMSprop(generator_w.parameters(), lr=0.00005)
d_optimizer_w = optim.RMSprop(discriminator_w.parameters(), lr=0.00005)
```

```
generator_gp = Generator(100).to(device)
discriminator_gp = DiscriminatorW().to(device)
g_optimizer_gp = optim.Adam(generator_gp.parameters(), lr=0.0002, betas=(0.5, 0.999))
d_optimizer_gp = optim.Adam(discriminator_gp.parameters(), lr=0.0002, betas=(0.5, 0.999))
```

```
train_gan(ls_generator, ls_discriminator, g_optimizer_ls, d_optimizer_ls, nn.MSELoss(), 50, "LS-GAN")
```

```
train_gan(generator_w, discriminator_w, g_optimizer_w, d_optimizer_w, None, 50, "WGAN")
```

```
train_gan(generator_gp, discriminator_gp, g_optimizer_gp, d_optimizer_gp, None, 50, "WGAN-GP")
```

```
import torch
from torchmetrics.image.inception import InceptionScore
from torchmetrics.image.fid import FrechetInceptionDistance
from torchvision.utils import make_grid
from torch.utils.tensorboard import SummaryWriter

def evaluate_gan(generator, num_images=100, latent_dim=100, device='cuda', log_dir="logs"):
    generator.eval()
    generator.to(device)

    writer = SummaryWriter(log_dir)  # Initialize TensorBoard writer
    fake_images = []

    with torch.no_grad():
        for _ in range(num_images):
            z = torch.randn(1, latent_dim, device=device)
            generated = generator(z)

            if generated.dim() == 3:
                generated = generated.unsqueeze(0)
            elif generated.dim() == 2:
                generated = generated.unsqueeze(0).unsqueeze(0)

            if generated.shape[1] == 1:
                generated = generated.expand(-1, 3, -1, -1)

            fake_images.append(generated.cpu())

    fake_images = torch.cat(fake_images, dim=0)

    # Normalize images for visualization
    fake_images = (fake_images - fake_images.min()) / (fake_images.max() - fake_images.min())

    # Convert to uint8 for metrics
    fake_images_uint8 = (fake_images * 255).clamp(0, 255).byte()

    # Log generated images to TensorBoard
    img_grid = make_grid(fake_images[:25], nrow=5)  # Show first 25 images in a 5x5 grid
    writer.add_image("Generated Images", img_grid, 0)

    # Compute Inception Score
    is_metric = InceptionScore().to(device)
    is_mean, is_std = is_metric(fake_images_uint8.to(device))
    is_score = is_mean.item()
    writer.add_scalar("Inception Score", is_score, 0)

    # Compute FID Score
    fid_metric = FrechetInceptionDistance().to(device)
    fid_metric.update(fake_images_uint8.to(device), real=False)
    fid_metric.update(fake_images_uint8.to(device), real=True)  # Replace with real images
    fid_score = fid_metric.compute().item()
    writer.add_scalar("FID Score", fid_score, 0)

    writer.close()  # Close TensorBoard writer

    print(f"Inception Score: {is_score:.4f} ± {is_std.item():.4f}")
    print(f"FID Score: {fid_score:.4f}")

    return is_score, fid_score
```

```
evaluate_gan(ls_generator)
```

```
c:\Users\Simrann\anaconda3\Lib\site-packages\torchmetrics\utilities\prints.py:43: UserWarning: Metric `InceptionScore` will save all ext
  warnings.warn(*args, **kwargs)
Inception Score: 1.0008 ± 0.0003
FID Score: -0.0000
(1.0008093118667603, -1.3069113720121095e-06)
```

```
evaluate_gan(generator_w)
```

```
Inception Score: 1.0930 ± 0.0430
FID Score: -0.0000
(1.092960238456726, -3.7507892557187006e-05)
```

```
evaluate_gan(generator_gp)
```

```
Inception Score: 1.0000 ± 0.0000
FID Score: -0.0000
(1.0, -1.1001999311588406e-10)
```