گزارش پروژه پیاده‌سازی شبکه‌های عصبی مولد تخاصمی (GAN) با استفاده از PyTorch

در این پروژه، ما به طراحی، پیاده‌سازی و آموزش یک شبکه عصبی مولد تخاصمی (GAN) می‌پردازیم که قادر است تصاویر دیجیتالی دست‌نویس را تولید کند. GANها از دو شبکه عصبی تشکیل شده‌اند: یک دیسکریمیناتور که وظیفه‌ی تشخیص تصاویر واقعی از تصاویر تولیدی را دارد و یک جنریتور که وظیفه‌ی تولید تصاویر جدید و شبه‌واقعی را بر عهده دارد. این دو شبکه در یک محیط رقابتی یاد می‌گیرند تا عملکرد یکدیگر را به چالش بکشند.

تنظیمات اولیه و وارد کردن کتابخانه‌ها:

کد زیر کتابخانه‌های مورد نیاز برای پروژه را وارد می‌کند:

```
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor
from torchvision.utils import make_grid
import matplotlib.pyplot as plt

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

torch و torchvision برای ساخت مدل‌ها و کار با داده‌ها استفاده می‌شود.

matplotlib برای نمایش تصاویر استفاده می‌شود.

تعریف کلاس دیسکریمیناتور:

دیسکریمیناتور با استفاده از معماری زیر ساخته شده است:

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.network = nn.Sequential(
            nn.Conv2d(1, 64, 5, stride=2, padding=2),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),
            nn.Conv2d(64, 128, 5, stride=2, padding=2),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),
            nn.Flatten(),
            nn.Linear(128*7*7, 1),
            nn.Sigmoid(),
        )

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

این دیسکریمیناتور از لایه‌های کانولوشنی و فعال‌ساز LeakyReLU استفاده می‌کند تا ویژگی‌های تصاویر ورودی را استخراج کند. Dropout برای جلوگیری از بیش‌برازش و Sigmoid برای تشخیص نهایی واقعی یا جعلی بودن تصاویر به کار رفته است.

تعریف کلاس جنریتور:

جنریتور برای تولید تصاویر جدید از نویز تصادفی طراحی شده است:

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.network = nn.Sequential(
            nn.ConvTranspose2d(100, 128, 7, stride=1,
padding=0),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.ConvTranspose2d(128, 64, 4, stride=2, padding=1),
            nn.BatchNorm2d(64),
```

```
            nn.ReLU(),
            nn.ConvTranspose2d(64, 1, 4, stride=2, padding=1),
            nn.Tanh(),  # Normalize MNIST images to range [-1,
1]
        )

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

این جنریتور با استفاده از لایه‌های ConvTranspose برای توسعه بعدی تصاویر، BatchNorm برای تثبیت یادگیری، و ReLU برای فعال‌سازی غیرخطی عمل می‌کند. Tanh برای نرمال‌سازی خروجی‌ها به بازه [-1, 1] استفاده می‌شود، که مطابق با فرمت داده‌های MNIST است.

مراحل آموزش و نمایش داده‌ها در پروژه GAN

آموزش دیسکریمیناتور و جنریتور:

در پروژه‌ی GAN، دو مرحله اصلی آموزش وجود دارد: آموزش دیسکریمیناتور و آموزش جنریتور. این دو مرحله به صورت متناوب انجام می‌شود تا اطمینان حاصل شود که هر دو مدل به طور موثری یاد می‌گیرند و در نهایت توانایی تشخیص و تولید تصاویر واقع‌بینانه‌تر را پیدا می‌کنند.

آموزش دیسکریمیناتور:

دیسکریمیناتور با دریافت تصاویر واقعی و تصاویر تولید شده توسط جنریتور آموزش می‌بیند. هدف از آموزش دیسکریمیناتور این است که بتواند بین تصاویر واقعی و جعلی تفاوت قائل شود. تصاویر واقعی با برچسب "۱" و تصاویر جعلی با برچسب "۰" مشخص می‌شوند. دیسکریمیناتور بر اساس خروجی‌های خود و با استفاده از تابع خطا، بهینه‌سازی می‌شود.

آموزش جنریتور:

جنریتور بر اساس نویز تصادفی که به عنوان ورودی دریافت می‌کند، تصاویر جعلی تولید می‌کند. هدف از آموزش جنریتور این است که تصاویر تولیدی آن به قدری واقع‌بینانه باشند که دیسکریمیناتور نتواند آن‌ها را از تصاویر واقعی تشخیص دهد. جنریتور بر اساس میزان توانایی فریب دیسکریمیناتور بهینه‌سازی می‌شود، به طوری که سعی می‌کند خطای دیسکریمیناتور را افزایش دهد.

```
def train_discriminator(discriminator, real_images, fake_images,
real_labels, fake_labels, optimizer_d, criterion):
    discriminator.zero_grad()
    outputs = discriminator(real_images)
    d_loss_real = criterion(outputs, real_labels)
    outputs = discriminator(fake_images.detach())
    d_loss_fake = criterion(outputs, fake_labels)
    d_loss = d_loss_real + d_loss_fake
    d_loss.backward()
    optimizer_d.step()
    return d_loss.item(), outputs.mean().item()

def train_generator(generator, discriminator, fake_images,
real_labels, optimizer_g, criterion):
    generator.zero_grad()
    outputs = discriminator(fake_images)
    g_loss = criterion(outputs, real_labels)
    g_loss.backward()
    optimizer_g.step()
    return g_loss.item()
```

نمایش داده‌ها:

در پایان هر دور از آموزش، تصاویر تولید شده توسط جنریتور به منظور بررسی پیشرفت و ارزیابی کیفیت تصاویر جعلی نمایش داده می‌شوند. این کار با استفاده از تابع show_images انجام می‌شود که تصاویر را به صورت یک گرید نمایش می‌دهد. تصاویر به فرمت مناسب برای نمایش تبدیل شده و با استفاده از کتابخانه matplotlib نمایش داده می‌شوند. این نمایش به توسعه‌دهنده امکان می‌دهد تا میزان واقع‌بینانه بودن تصاویر تولیدی را ارزیابی کند و در صورت نیاز تنظیمات مدل را تغییر دهد.

```
def show_images(images, num_images=25):
    images = (images + 1) / 2  # Rescale images to range [0, 1]
    grid = make_grid(images[:num_images], nrow=5)
    plt.figure(figsize=(5, 5))
    plt.imshow(grid.permute(1, 2, 0).cpu().numpy(), cmap='gray')
    plt.axis('off')
    plt.show()
```

```
# Hyperparameters
lr = 0.0002
batch_size = 64
```

```
epochs = 50
pretrain_epochs = 0  # Pre-train the discriminator

# Data loading
train_data = datasets.MNIST(root="data", train=True,
download=True, transform=ToTensor())
train_loader = DataLoader(train_data, batch_size=batch_size,
shuffle=True)
```

```
# Initialize models and move to device
discriminator = Discriminator().to(device)
generator = Generator().to(device)

# Optimizers
optimizer_d = torch.optim.Adam(discriminator.parameters(),
lr=lr)
optimizer_g = torch.optim.Adam(generator.parameters(), lr=lr)

# Loss function
criterion = nn.BCELoss()

# Pre-train the discriminator
for epoch in range(pretrain_epochs):
    for i, (real_images, _) in enumerate(train_loader):
        real_images = real_images.to(device)
        real_labels = torch.ones(real_images.size(0),
1).to(device)
        fake_labels = torch.zeros(real_images.size(0),
1).to(device)

        noise = torch.randn(real_images.size(0), 100, 1,
1).to(device)
        fake_images = generator(noise)

        # Train discriminator on real and generated data
        d_loss, _ = train_discriminator(discriminator,
real_images, fake_images, real_labels, fake_labels, optimizer_d,
criterion)

        if (i + 1) % 200 == 0:
            print(f'Pre-Training Epoch [{epoch +
1}/{pretrain_epochs}], Step [{i + 1}/{len(train_loader)}],
d_loss: {d_loss:.4f}')
```

```
# Normal training loop
for epoch in range(epochs):
    for i, (real_images, _) in enumerate(train_loader):
        real_images = real_images.to(device)
        real_labels = torch.ones(real_images.size(0),
1).to(device)
        fake_labels = torch.zeros(real_images.size(0),
1).to(device)

        # Generate fake images
        noise = torch.randn(real_images.size(0), 100, 1,
1).to(device)
        fake_images = generator(noise)

        # Train discriminator
        d_loss, d_real_score =
train_discriminator(discriminator, real_images, fake_images,
real_labels, fake_labels, optimizer_d, criterion)

        # Train generator
        g_loss = train_generator(generator, discriminator,
fake_images, real_labels, optimizer_g, criterion)

        # Logging the process
        if (i + 1) % 200 == 0:
            print(f'Epoch [{epoch + 1}/{epochs}], Step [{i +
1}/{len(train_loader)}], d_loss: {d_loss:.4f}, g_loss:
{g_loss:.4f}, D(real): {d_real_score:.2f}')

    # Displaying generated images at the end of each epoch
    with torch.no_grad():
        test_noise = torch.randn(batch_size, 100, 1,
1).to(device)
        generated_images = generator(test_noise)
        show_images(generated_images)
        print(f'Generated images from Epoch {epoch + 1}')
```

تعریف پارامترهای آموزشی:

lr: نرخ یادگیری برای بهینه‌سازها، تعیین‌کننده سرعت به‌روزرسانی وزن‌ها در شبکه.

batch_size: تعداد تصاویر در هر دسته از داده‌ها که در یک مرحله آموزشی مورد استفاده قرار می‌گیرد.

epochs: تعداد دوره‌های کامل آموزش بر روی کل داده‌های آموزشی.

pretrain_epochs: تعداد دوره‌های آموزش اولیه برای دیسکریمیناتور قبل از شروع آموزش تناوبی با جنریتور.

بارگذاری داده‌ها:

داده‌های MNIST شامل تصاویر دست‌نویس اعداد از 0 تا 9، که برای آموزش مدل‌ها استفاده می‌شوند.

مقداردهی اولیه مدل‌ها و تنظیم دستگاه:

discriminator و generator مدل‌های مورد نظر برای دیسکریمیناتور و جنریتور هستند.

این مدل‌ها به دستگاه محاسباتی مورد نظر انتقال داده می‌شوند (معمولا GPU).

تعریف بهینه‌سازها و تابع خسارت:

بهینه‌سازهای مجزا برای دیسکریمیناتور و جنریتور با نرخ یادگیری مشخص.

تابع خسارت (BCELoss) برای محاسبه خسارت بر اساس مقایسه خروجی‌های مدل با برچسب‌های واقعی.

آموزش دیسکریمیناتور و جنریتور:

دوره‌های آموزش اولیه برای دیسکریمیناتور به منظور بهبود توانایی تشخیص تصاویر واقعی از جعلی.

دوره‌های آموزشی تناوبی بین دیسکریمیناتور و جنریتور برای بهبود تولید تصاویر جعلی واقع‌بینانه و همچنین بهبود توانایی تشخیص دیسکریمیناتور.

نمایش تصاویر تولید شده:

در پایان هر دوره آموزشی، تصاویر تولید شده توسط جنریتور با استفاده از تابع show_images نمایش داده می‌شوند تا پیشرفت جنریتور را بصری سنجید.

این بخش از کد نشان‌دهنده ساختار کلی و فرآیند آموزش در پروژه‌های GAN است و تمام اجزای مورد نیاز برای آموزش موثر شبکه‌های عصبی مولد تخاصمی را در بر می‌گیرد.

این دو خط کد وضعیت آموزشی دو مدل generator و discriminator را در فایل‌هایی با نام‌های generator.pth و discriminator.pth ذخیره می‌کنند. این امکان اجازه می‌دهد که مدل‌ها بدون نیاز به آموزش مجدد، در آینده قابل استفاده و بررسی باشند.

```
# Save the model checkpoints
torch.save(generator.state_dict(), 'generator.pth')
torch.save(discriminator.state_dict(), 'discriminator.pth')
```

یک نمونه از خروجی در ایپاک های پایین: (به دلیل محدودیت دسترسی به جی پی یو و کمبود وقت نشد کامل ران بگیرم و عکس تولید شده نهایی رو قرار بدم)