实 验 报 告

课程名称	内容安全				
学生姓名		学号		指导老师	**崔老师**
专业	**网络空间安全**	班级		实验时间	

一、实验内容

- 采用 DCGAN 或 WGAN 或 WGAN-GP 生成 1000 张不同的二次元漫画脸图像

二、实验原理

1、生成对抗网络（GAN）

生成对抗网络(Generative Adversarial Network, GAN)是由 Ian Goodfellow 等人在 2014 年提出的一种深度学习模型架构。GAN 由两个神经网络组成：生成器(Generator)和判别器(Discriminator)，它们通过一个对抗过程相互博弈训练。

GAN 的核心思想是建立一个博弈论场景，其中生成器网络 G 试图生成逼真的样本以欺骗判别器网络 D，而判别器网络则尝试区分真实样本和生成样本。这两个网络在训练过程中相互对抗、相互促进，最终生成器能够生成高质量的逼真样本。

数学上，GAN 的目标函数可以表示为一个极小极大博弈：

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log(D(x))] + \mathbb{E}_{z\sim p_z(z)}[\log(1-D(G(z)))]$$

其中，G 表示生成器（Generator），它的作用是将随机噪声 z（一般从某个先验分布 $p_z(z)$ 中采样得到）转换为生成的数据 G(z)，目的是尽量生成与真实数据相似的数据。

D 表示判别器（Discriminator），它的输入是真实数据 x 或生成器生成的数据 G(z)，输出是一个标量，表示输入数据为真实数据的概率，其目标是尽可能准

确地区分真实数据和生成数据。

$V(D,G)$ 是价值函数，用于衡量生成器和判别器的性能。

$p_{data}(x)$ 表示真实数据分布

$p_z(z)$ 表示潜在空间噪声分布

$G(z)$ 生成器将潜在空间向量映射到数据空间

$D(z)$ 判别器输出实例 xx x 为真实样本的概率

然而，标准 GAN 在训练过程中存在诸多问题，如模式崩溃(mode collapse)、训练不稳定、梯度消失等，这些问题在实践中限制了 GAN 的应用。

2、Wasserstein GAN (WGAN)

为了解决原始 GAN 的问题，Arjovsky 等人在 2017 年提出了 Wasserstein GAN (WGAN)。WGAN 的核心创新在于将判别器(Discriminator)替换为评论家(Critic)，并采用 Wasserstein 距离（也称 Earth Mover's Distance，EMD）来度量真实分布和生成分布之间的差异，而不是 JS 散度。

Wasserstein 距离可以提供一个平滑的梯度，即使在生成分布和真实分布没有重叠或者重叠很少的情况下也是如此。这解决了原始 GAN 中的梯度消失问题。

WGAN 的目标函数可以表示为：

$$\min_G \max_{D \in \mathcal{D}} \mathbb{E}_{x \sim p_{data}(x)}[D(x)] - \mathbb{E}_{z \sim p_z(z)}[D(G(z))]$$

其中 D 是所有 1-Lipschitz 连续函数的集合。为了确保 Critic 函数满足 Lipschitz 连续性，WGAN 引入了权重裁剪(weight clipping)技术，将网络参数限制在一个紧凑的空间中。

3、Wasserstein GAN with Gradient Penalty (WGAN-GP)

虽然 WGAN 改进了原始 GAN 的训练稳定性，但权重裁剪可能导致模型容量受限和梯度消失/爆炸等问题。为了解决这些问题，Gulrajani 等人在 2017 年提出了 WGAN-GP，即带梯度惩罚的 Wasserstein GAN。

WGAN-GP 通过在目标函数中添加梯度惩罚项来替代权重裁剪，以更好地强制执行 Lipschitz 约束。具体来说，它惩罚真实样本和生成样本之间随机插值点的梯度范数偏离 1 的情况。

WGAN-GP 的目标函数可以表示为：

$$\min_G \max_D \left\{ \mathbb{E}_{x\sim p_{data}(x)}[D(x)] - \mathbb{E}_{z\sim p_z(z)}[D(G(z))] - \lambda \mathbb{E}_{\hat{x}\sim p_{\hat{x}}(\hat{x})} \left[(\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1)^2 \right] \right\}$$

其中，λ 为梯度惩罚的权重系数；

$\hat{x}$ 表示真实样本和生成样本之间的插值点，$\hat{x} = \varepsilon x + (1-\varepsilon)G(z), \varepsilon \sim U[0,1]$

$p_{\hat{x}}$ 为插值点的分布。

WGAN-GP 相比原始 WGAN，具有以下优势：

- 不需要权重裁剪，避免了模型容量受限问题
- 收敛性更好，训练更稳定
- 生成样本质量更高
- 网络架构选择更灵活，不需要避免使用 batch normalization 等技术

因此，本实验选择使用 WGAN-GP 来生成二次元漫画脸图像，以获得更稳定的训练过程和更高质量的生成结果。

三、实验步骤

1、数据集处理（见 dataset.py）

AnimeDataset 类继承自 PyTorch 的 Dataset 类

首先初始化函数接收数据根目录路径和可选的图像变换函数

通过遍历目录获取所有有效图像文件，支持.jpg、.jpeg、.png、.bmp 格式

如果指定目录不存在，将使用空列表，后续会生成随机噪声图像用于调试

```
    class AnimeDataset(Dataset):
        def __init__(self, root_dir, transform=None):
            """
                root_dir (string): 包含图像的目录路径
                transform (callable, optional): 应用于样本的可选变换
            """
            self.root_dir = root_dir
            self.transform = transform

            # 获取所有图像文件
            self.image_files = []
            valid_extensions = {'.jpg', '.jpeg', '.png', '.bmp'}

            # 遍历目录获取所有有效图像文件
            if os.path.exists(root_dir):
```

```
            for file in os.listdir(root_dir):
                ext = os.path.splitext(file)[1].lower()
                if ext in valid_extensions:
                    self.image_files.append(os.path.join(root_dir, file))
        else:
            print(f"警告: 目录 {root_dir} 不存在")
            self.image_files = []  # 如果目录不存在，则使用空列表

        print(f"找到 {len(self.image_files)} 个图像文件。")
```

__len__方法返回数据集中样本的数量，至少为 1，确保即使没有图像文件也能进行训练。

__getitem__方法负责获取指定索引的样本：

正常情况下，加载索引对应的图像文件并转换为 RGB 模式；如果没有图像文件，生成 64×64 的随机 RGB 噪声图像；如果图像加载失败，同样使用随机噪声代替；最后应用指定的变换（如调整大小、标准化等）并返回处理后的图像张量。

```
    def __len__(self):
        """返回数据集中样本的数量"""
        # 至少返回1，即使没有图像文件，以便进行训练
        return max(1, len(self.image_files))

    def __getitem__(self, idx):
        """
            idx (int): 样本索引
        """
        # 如果没有图像文件，则返回随机噪声（调试用）
        if len(self.image_files) == 0:
            # 创建随机噪声图像
            import numpy as np
            random_image = np.random.rand(64, 64, 3) * 255
            image = Image.fromarray(random_image.astype('uint8'))
        else:
            # 正常情况：加载图像文件
            img_path = self.image_files[idx % len(self.image_files)]
            try:
                image = Image.open(img_path).convert('RGB')
            except Exception as e:
                print(f"无法加载图像 {img_path}: {e}")
                # 加载失败时使用随机噪声代替
                import numpy as np
                random_image = np.random.rand(64, 64, 3) * 255
```

```
                image = Image.fromarray(random_image.astype('uint8'))

        # 应用变换
        if self.transform:
            image = self.transform(image)

        return image
```

2、模型架构（见 models.py）

（1）生成器网络

生成器网络的设计遵循典型的 GAN 生成器架构，主要由两部分组成：

- 全连接层

  将低维潜在空间向量映射到高维特征空间。这里 init_size 是最终图像尺寸的 1/4。

- 卷积块

  负责将特征映射逐步上采样到目标图像尺寸。主要包含：

  - 批量归一化层：稳定训练过程，加速收敛
  - 上采样层：将特征图尺寸放大，共进行两次上采样，每次放大 2 倍，将 16×16 特征图变为 64×64
  - 卷积层：增强模型表达能力，捕捉空间特征
  - LeakyReLU 激活函数：引入非线性，避免传统 ReLU 的"死亡"问题
  - Tanh 激活函数：最后一层使用 Tanh 将输出限制在[-1,1]范围内，与标准化的输入图像范围匹配

```
class Generator(nn.Module):
    def __init__(self, latent_dim, channels=3, img_size=64):
        super(Generator, self).__init__()
        self.img_size = img_size
        self.latent_dim = latent_dim
        self.channels = channels

        # 计算初始特征图的尺寸
        self.init_size = self.img_size // 4  # 最终图像尺寸的1/4

        # 第一层：将潜在向量映射到特征图
        self.l1 = nn.Sequential(
            nn.Linear(latent_dim, 128 * self.init_size ** 2)
```

```
        )

        # 卷积块：逐步上采样到目标图像尺寸
        self.conv_blocks = nn.Sequential(
            nn.BatchNorm2d(128),
            nn.Upsample(scale_factor=2),
            nn.Conv2d(128, 128, 3, stride=1, padding=1),
            nn.BatchNorm2d(128, 0.8),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Upsample(scale_factor=2),
            nn.Conv2d(128, 64, 3, stride=1, padding=1),
            nn.BatchNorm2d(64, 0.8),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, channels, 3, stride=1, padding=1),
            nn.Tanh(),
        )
```

前向传播

- 输入的潜在向量 z 首先通过全连接层映射到高维特征空间
- 然后将一维特征向量重塑为三维特征图(batch_size × 128 × init_size × init_size)
- 最后经过卷积块处理，生成最终的图像(batch_size × channels × img_size × img_size)

```
    def forward(self, z):
        # 将潜在向量映射到特征空间
        out = self.l1(z)
        # 重塑为卷积特征图
        out = out.view(out.shape[0], 128, self.init_size, self.init_size)
        # 应用卷积块生成图像
        img = self.conv_blocks(out)
        return img
```

（2）评论家网络

评论家网络(Critic)的结构与传统判别器类似，但输出不经过 sigmoid 激活，而是直接输出一个实数值，用于估计 Wasserstein 距离。

- 基本判别器模块（discriminator_block），包含卷积层、非线性激活层、批量归一化层。
- 主体结构(self.model)由 4 个判别器块组成，逐步将通道数从输入的 3 或 1(灰度图)增加到 128，同时将图像尺寸缩小为原来的 1/16。

- 输出层(self.adv_layer)全连接层将特征映射为单一标量值，该值表示输入图像的"真实性得分"。

```
    class Critic(nn.Module):
        def __init__(self, channels=3, img_size=64):
            super(Critic, self).__init__()

            # 基本判别器
            def discriminator_block(in_filters, out_filters, bn=True):
                block = [nn.Conv2d(in_filters, out_filters, 3, 2, 1), nn.LeakyReLU(0.2, inplace=True)]
                if bn:
                    block.append(nn.BatchNorm2d(out_filters, 0.8))
                return block

            # 卷积层序列
            self.model = nn.Sequential(
                *discriminator_block(channels, 16, bn=False),
                *discriminator_block(16, 32),
                *discriminator_block(32, 64),
                *discriminator_block(64, 128),
            )

            # 计算卷积层输出的特征图大小
            ds_size = img_size // 2 ** 4  # 经过4次下采样

            # 全连接层输出单一值
            self.adv_layer = nn.Linear(128 * ds_size ** 2, 1)
```

评论家网络的前向传播函数

- 输入图像经过卷积层序列提取特征
- 将多维特征图展平为一维向量
- 通过全连接层输出单一实数值，表示输入图像的真实程度

```
        def forward(self, img):
            features = self.model(img)
            features = features.view(features.shape[0], -1)
            validity = self.adv_layer(features)
            return validity
```

3、工具函数（见utils.py）

weights_init_normal 函数用于初始化网络权重，这是 GAN 训练稳定性的关键因素：

- 对于卷积层，权重初始化为均值为 0，标准差为 0.02 的正态分布
- 对于批量归一化层，权重初始化为均值为 1，标准差为 0.02 的正态分布，偏置初始化为 0

这种初始化方式有助于训练的稳定性。

```
def weights_init_normal(m):
    """
    初始化模型权重
        m: 模型层
    """
    classname = m.__class__.__name__
    if classname.find("Conv") != -1:
        torch.nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find("BatchNorm2d") != -1:
        torch.nn.init.normal_(m.weight.data, 1.0, 0.02)
        torch.nn.init.constant_(m.bias.data, 0.0)
```

save_images 函数负责保存生成的图像

- 使用 torchvision 的 make_grid 函数将批次图像拼接成网格
- 保存网格图像，文件名包含当前轮数和批次索引
- 对于最终生成的图像会单独保存

```
def save_images(images, path, epoch, batch_i, normalize=True):
    """
    保存生成的图像
        images: 生成的图像批次
        path: 保存路径
        epoch: 当前训练轮数
        batch_i: 当前批次索引
        normalize: 是否归一化图像
    """
    # 确保目录存在
    os.makedirs(path, exist_ok=True)

    # 保存整个批次的图像网格
    grid = vutils.make_grid(images, padding=2, normalize=normalize)
    save_path = os.path.join(path, f"epoch_{epoch}_batch_{batch_i}.png")
    vutils.save_image(grid, save_path, normalize=normalize)
```

```
17.
18.         # 单独保存每张图像
19.         if "final" in path:  # 只为最终生成的图像单独保存
20.             img_dir = os.path.join(path, "individual")
21.             os.makedirs(img_dir, exist_ok=True)
22.
23.             for i, img in enumerate(images):
24.                 # 计算全局索引，确保文件名唯一
25.                 global_idx = batch_i * images.shape[0] + i
26.                 img_path = os.path.join(img_dir, f"image_{global_idx:04d}.png")
27.                 vutils.save_image(img, img_path, normalize=normalize)
```

4、训练流程（见 mian.py）

设置超参数，包括训练的相关参数（训练轮数、批次大小、学习率）、模型的相关参数（潜在空间维度、图像大小、图像通道数、评论家训练次数、梯度惩罚权重）、路径参数（数据集路径、输出路径、模型保存路径）

```
1.  parser = argparse.ArgumentParser()
2.  parser.add_argument("--n_epochs", type=int, default=50, help="训练的轮数")
3.  parser.add_argument("--batch_size", type=int, default=64, help="批次大小")
4.  parser.add_argument("--lr", type=float, default=0.0002, help="学习率")
5.  parser.add_argument("--latent_dim", type=int, default=100, help="潜在空间维度")
6.  parser.add_argument("--img_size", type=int, default=64, help="图像大小")
7.  parser.add_argument("--channels", type=int, default=3, help="图像通道数")
8.  parser.add_argument("--n_critic", type=int, default=5, help="critic 训练次数")
9.  parser.add_argument("--sample_interval", type=int, default=400, help="保存样本的间隔")
10. parser.add_argument("--lambda_gp", type=float, default=10, help="梯度惩罚项的权重")
11. parser.add_argument("--dataset_path", type=str, default="data/anime_faces", help="数据集路径")
12. parser.add_argument("--output_path", type=str, default="generated_images", help="输出路径")
13. parser.add_argument("--model_path", type=str, default="saved_models", help="模型保存路径")
14. opt = parser.parse_args()
```

准备程序的训练环境

- 创建必要的输出目录
- 设置计算设备
- 初始化生成器和评论家网络，并将其移动到指定设备
- 应用权重初始化函数

```
1.  # 创建输出目录
2.  os.makedirs(opt.output_path, exist_ok=True)
```

```
    os.makedirs(opt.model_path, exist_ok=True)

    device = torch.device("cpu")
    print(f"使用设备: {device}")

    # 初始化生成器和判别器
    generator = Generator(opt.latent_dim, opt.channels, opt.img_size).to(device)
    critic = Critic(opt.channels, opt.img_size).to(device)

    # 初始化权重
    generator.apply(weights_init_normal)
    critic.apply(weights_init_normal)
```

设置数据处理和优化器：

- 定义图像预处理流程。调整大小、转换为张量、标准化到[-1,1]范围
- 加载数据集并创建数据加载器，设置批次大小、随机打乱、丢弃不完整批次(drop_last=True)
- 为生成器和评论家创建 Adam 优化器，使用推荐的 GAN 超参数(betas=(0.5, 0.9))

```
    # 配置数据预处理
    transforms_list = [
        transforms.Resize((opt.img_size, opt.img_size)),
        transforms.ToTensor(),
        transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5]),
    ]
    transform = transforms.Compose(transforms_list)

    # 加载数据集
    dataset = AnimeDataset(opt.dataset_path, transform=transform)
    dataloader = DataLoader(
        dataset,
        batch_size=opt.batch_size,
        shuffle=True,
        num_workers=0,
        drop_last=True,
    )

    # 优化器
    optimizer_G = optim.Adam(generator.parameters(), lr=opt.lr, betas=(0.5, 0.9))
    optimizer_C = optim.Adam(critic.parameters(), lr=opt.lr, betas=(0.5, 0.9))
```

梯度惩罚：

首先生成随机权重 alpha，利用 alpha 创建真实样本和生成样本之间的线性插值点；计算评论家对插值样本的输出；使用 PyTorch 的自动求导功能计算梯度；计算梯度的 L2 范数，并惩罚其与 1 的偏差平方。

```
    def compute_gradient_penalty(critic, real_samples, fake_samples):
        """计算 WGAN-GP 中的梯度惩罚项"""
        # 随机数 alpha
        alpha = torch.rand(real_samples.size(0), 1, 1, 1, device=device)
        # 在真实样本和生成样本之间进行线性插值
        interpolates = (alpha * real_samples + ((1 - alpha) * fake_samples)).requires_grad_(True)
        # 计算判别器对插值样本的输出
        d_interpolates = critic(interpolates)
        # 创建一个全 1 的张量
        fake = torch.ones(d_interpolates.size(), device=device, requires_grad=False)
        # 计算梯度
        gradients = torch.autograd.grad(
            outputs=d_interpolates,
            inputs=interpolates,
            grad_outputs=fake,
            create_graph=True,
            retain_graph=True,
            only_inputs=True,
        )[0]
        # 计算梯度的范数
        gradients = gradients.view(gradients.size(0), -1)
        gradient_penalty = ((gradients.norm(2, dim=1) - 1) ** 2).mean()
        return gradient_penalty
```

训练评论家网络：

- 将真实图像移动到指定设备
- 生成随机噪声向量，并用生成器生成假图像
- 计算评论家对真实图像和假图像的评估得分
- 计算梯度惩罚
- 计算 WGAN-GP 评论家损失：真实样本得分的负均值+假样本得分的正均值+加权梯度惩罚
- 反向传播并更新评论家参数

```
    batches_done = 0
    for epoch in range(opt.n_epochs):
        for i, imgs in enumerate(tqdm(dataloader, desc=f"Epoch {epoch}")):
```

```
            # 真实图像
            real_imgs = imgs.to(device)
            batch_size = real_imgs.size(0)

            #  训练 Critic
            optimizer_C.zero_grad()
            # 生成随机噪声
            z = torch.randn(batch_size, opt.latent_dim, device=device)
            # 生成假图像
            fake_imgs = generator(z)

            # 计算真实图像和假图像的 Critic 输出
            real_validity = critic(real_imgs)
            fake_validity = critic(fake_imgs.detach())

            # 计算梯度惩罚
            gradient_penalty = compute_gradient_penalty(critic, real_imgs, fake_imgs.detach())

            # Critic 损失函数
            d_loss = -torch.mean(real_validity) + torch.mean(fake_validity) + opt.lambda_gp * gradient
    _penalty

            d_loss.backward()
            optimizer_C.step()
```

训练生成器网络，在评论家完成 n_critic 次训练后执行

- 生成新的随机噪声并创建假图像
- 计算评论家对新生成图像的评估得分
- 计算生成器损失：评论家对假样本得分的负均值
- 反向传播并更新生成器参数
- 打印当前训练状态，包括批次信息和损失值

生成器的目标是最大化评论家对假样本的评分，相当于最小化假样本。

```
    # 只有在 critic 完成 n_critic 次训练后才训练 Generator
            if i % opt.n_critic == 0:
                #  训练 Generator
                optimizer_G.zero_grad()

                # 生成新的假图像
                z = torch.randn(batch_size, opt.latent_dim, device=device)
                fake_imgs = generator(z)
```

```
9.
10.             # 计算判别器对生成图像的输出
11.             fake_validity = critic(fake_imgs)
12.
13.             # Generator 损失函数
14.             g_loss = -torch.mean(fake_validity)
15.
16.             g_loss.backward()
17.             optimizer_G.step()
18.
19.             print(
20.                 f"[Epoch {epoch}/{opt.n_epochs}] [Batch {i}/{len(dataloader)}] "
21.                 f"[D loss: {d_loss.item():.4f}] [G loss: {g_loss.item():.4f}]"
22.             )
```

在每个训练轮次结束后，程序会保存生成器和评论家的模型权重

```
1.     torch.save(generator.state_dict(), os.path.join(opt.model_path, f"generator_epoch_{epoch}.pth"))
2.     torch.save(critic.state_dict(), os.path.join(opt.model_path, f"critic_epoch_{epoch}.pth"))
```

训练完成后，程序使用最终训练好的生成器模型生成 1000 张图像

```
1.   with torch.no_grad():
2.       generator.eval()
3.       num_generated = 0
4.       num_batches = (1000 + opt.batch_size - 1) // opt.batch_size  # 计算需要多少批次来生成 1000 张图像
5.
6.       for i in range(num_batches):
7.           # 计算这一批次要生成多少图像
8.           current_batch_size = min(opt.batch_size, 1000 - num_generated)
9.           if current_batch_size <= 0:
10.              break
11.          # 生成随机噪声
12.          z = torch.randn(current_batch_size, opt.latent_dim, device=device)
13.          # 生成图像
14.          fake_imgs = generator(z)
15.          # 保存图像
16.          save_images(fake_imgs, os.path.join(opt.output_path, "final"), 0, i, normalize=True)
17.          num_generated += current_batch_size
18.          print(f"已生成 {num_generated}/1000 张图像")
```

5、图像生成（generate_image.py）

首先同样通过 argparse 解析命令行参数，参数指定了要生成的图像数量、

潜在空间维度、图像大小、图像通道数、生成器模型路径、输出目录、生成批次大小。

```
    parser = argparse.ArgumentParser()
    parser.add_argument("--n_images", type=int, default=1000, help="要生成的图像数量")
    parser.add_argument("--latent_dim", type=int, default=100, help="潜在空间维度")
    parser.add_argument("--img_size", type=int, default=64, help="图像大小")
    parser.add_argument("--channels", type=int, default=3, help="图像通道数")
    parser.add_argument("--model_path", type=str, required=True, help="生成器模型路径")
    parser.add_argument("--output_dir", type=str, default="generated_images", help="输出目录")
    parser.add_argument("--batch_size", type=int, default=64, help="生成批次大小")
    opt = parser.parse_args()
```

准备环境并加载模型

```
    os.makedirs(opt.output_dir, exist_ok=True)

    # CPU 设备
    device = torch.device("cpu")
    print(f"使用设备: {device}")

    # 加载生成器模型
    generator = Generator(opt.latent_dim, opt.channels, opt.img_size).to(device)
    print(f"加载模型: {opt.model_path}")
    generator.load_state_dict(torch.load(opt.model_path, map_location=device))
    generator.eval()
```

批量生成并保存图像

计算需要的批次数，以生成指定数量的图像，在无梯度计算的上下文中进行推理，提高效率。

对每个批次：

计算当前批次应生成的图像数量；生成随机噪声并通过生成器创建图像；保存生成的图像并更新计数；显示生成进度

```
    print(f"开始生成 {opt.n_images} 张图像...")
    num_generated = 0
    num_batches = (opt.n_images + opt.batch_size - 1) // opt.batch_size

    with torch.no_grad():
        for i in range(num_batches):
            # 计算此批次要生成的图像数量
            current_batch_size = min(opt.batch_size, opt.n_images - num_generated)
```

```
            if current_batch_size <= 0:
                break

            # 生成随机噪声
            z = torch.randn(current_batch_size, opt.latent_dim, device=device)

            # 生成图像
            fake_imgs = generator(z)

            # 保存图像
            save_images(fake_imgs, opt.output_dir, 0, i, normalize=True)

            num_generated += current_batch_size
            print(f"已生成 {num_generated}/{opt.n_images} 张图像")

    print(f"已成功生成 {num_generated} 张图像，保存在 {opt.output_dir} 目录下。")
```

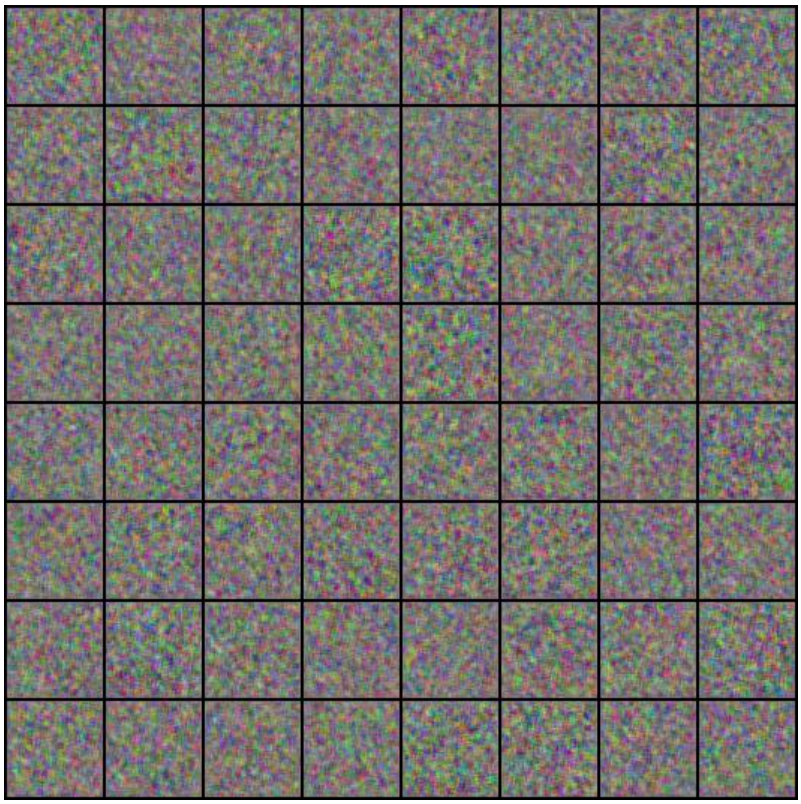
四、实验结果

由于本地计算资源限制，正式运行设置 epoch=50,batch_size=64，其余参数为代码中设定的默认参数。

训练集为 anime_faces，共有 21551 个图像文件

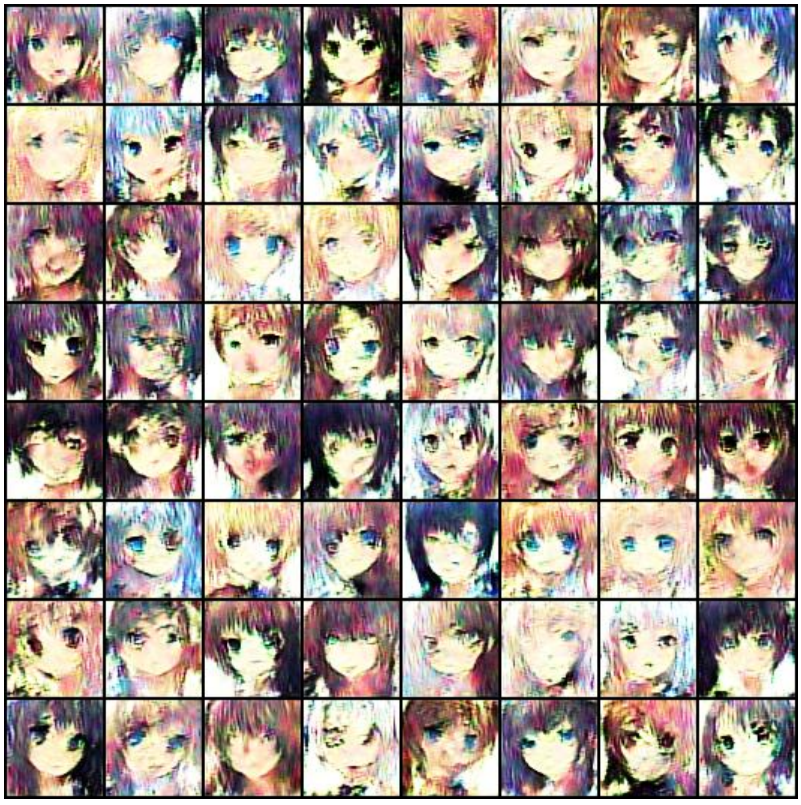
- 生成的 1000 张二次元漫画脸头像保存在 generated_images->final->individual
- 每一轮训练后生成的图像缩略图集锦保存在 generated_images 中
- 每一轮训练后的模型保存在 saved_models
- Epoch0~35 的详细训练情况、生成器和评论家损失率等详见“训练详细情况.docm”(由于本地资源限制，后面的 epoch 未能记录上，程序跑完后内存占用过量，电脑重启了 o(┬﹏┬)o）

Epoch=0 时生成的图像缩略图



Epoch=10 时生成的图像缩略图

Epoch=20 时生成的图像缩略图



Epoch=30 时生成的图像缩略图

Epoch=40 时生成的图像缩略图

Epoch=48 时生成的图像缩略图

可以看到，总体上来讲，随着训练轮数的增大，模型生成的图片更加细致和清晰。Epoch=10 的时候，图像已经有了基本的轮廓，但是可以看出一个个色块的感觉，清晰度不高。Epoch=20~30 的时候，清晰度有所改善，色块大小明显减小，epoch=40~50 的时候已经可以看出头发间颜色的分别，清晰度也有了大幅度提升，但部分图片的五官和某个别位置会出现混乱。

五、实验心得

本次实验中我完成了基于 WGAN-GP 实现二次元漫画脸生成器模型。我深入理解了 GAN、WGAN 和 WGAN-GP 的原理，了解了 WGAN 通过引入 Wasserstein 距离和评论家机制，有效解决梯度消失问题的原理，而 WGAN - GP 在此基础上，通过添加梯度惩罚项替代权重裁剪，进一步提升了训练的稳定性和生成样本的质量；并通过实际的模型构建让我对生成对抗网络有了更深刻、直观的认识。

在实验过程中，我掌握了使用 PyTorch 框架搭建深度学习模型的技能。学会了自定义数据集类、构建生成器和评论家网络结构、初始化网络权重、设置优化器以及实现训练和生成图像的流程。同时，还学会了如何处理数据，包括图像的读取、预处理和数据加载器的使用。这些技能对于今后开展其他深度学习相关的研究和项目具有重要的基础作用。

尽管本次实验成功生成了二次元漫画脸图像，但也存在一些不足之处。由于资源限制，遗失了部分模型性能指标，影响了对模型性能的全面评估。此外，生成的图片仍然会出现一些问题，包括清晰度仍不足，部分模块会出现混乱等现象。未来还可以对模型进行改进，尝试不同的网络结构、超参数设置，并使用更高算力的设备，以进一步提升生成图像的质量和多样性。