深度学习 Lab5 实验报告

PB19151769　　马宇骁

1　实验要求

理解 GAN 并选择一种 GAN 的算法以及相关数据集实现。

2　实验原理

2.1　GAN 生成对抗网络概述

GAN 包含有两个模型，一个是生成模型（generative model），一个是判别模型 (discriminative model)。

- 生成模型的任务是生成看起来自然真实的、和原始数据相似的实例。
- 判别模型的任务是判断给定的实例看起来是自然真实的还是人为伪造的（真实实例来源于数据集，伪造实例来源于生成模型）。

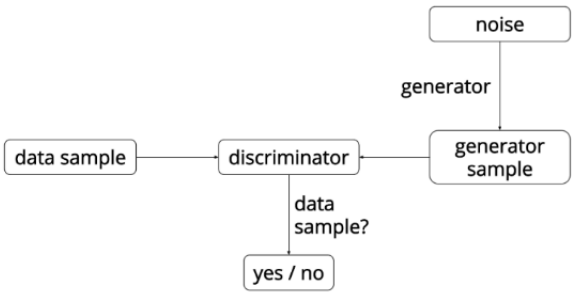

图 1: GAN

Generator 是一个生成图片的网络，它接收一个随机的噪声 z，通过这个噪声生成图片，记做 G(z)。Discriminator 是一个判别网络，判别一张图片是不是“真实的”。它的输入是 x，x 代表一张图片，输出 D（x）代表 x 为真实图片的概率，如果为 1，就代表 100% 是真实的图片，而输出为 0，就代表不可能是真实的图片。

2.2　训练

在训练过程中，生成网络的目标就是尽量生成真实的图片去欺骗判别网络 D。而网络 D 的目标就是尽量把网络 G 生成的图片和真实的图片分别开来。对于 GAN，情况就是生成模型 G 恢复了训练数据的分布（造出了和真实数据一模一样的样本），判别模型再也判别不出

来结果，准确率为 50%，约等于乱猜。这是双方网路都得到利益最大化，不再改变自己的策略，也就是不再更新自己的权重。

定义目标函数为：

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

即，G 网络的 loss 是 $\log(1-D(G(\boldsymbol{z}))$，而 D 的 loss 是 $-(\log(D(\boldsymbol{x})) + \log(1-D(G(\boldsymbol{z}))))$.

训练网络 D 使得最大化 D 的损失。而训练过程中固定一方，更新另一个网络的参数，交替迭代，使得对方的错误最大化，最终，G 能估测出样本数据的分布，也就是生成的样本更加的真实。G 网络的训练是希望 $D(G(\boldsymbol{z}))$ 趋近于 1，也就是正类，这样 G 的 loss 就会最小。而 D 网络的训练就是一个 2 分类，目标是分清楚真实数据和生成数据，也就是希望真实数据的 D 输出趋近于 1，而生成数据的输出即 $D(G(\boldsymbol{z}))$ 趋近于 0，或是负类。

3 实验实现

实验选择深度卷积生成对抗网络（DCGAN）（本来是选消除马赛克，尝试几次发现很有趣，但是因为要查重，文件太多从头重构压力太大，只修改部分函数和调用方式怕查重不过）生成动漫头像。

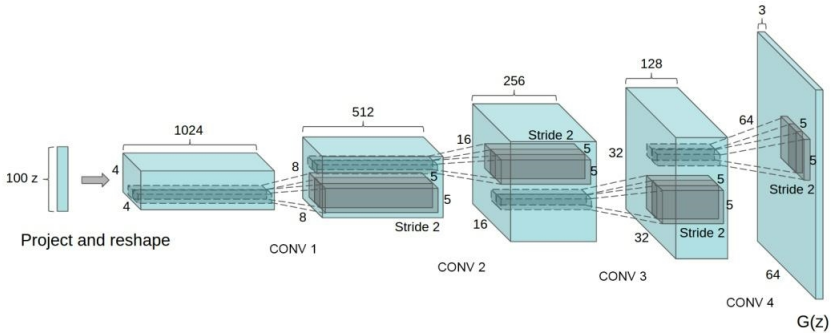

图 2: DCGAN

DCGAN 相对于原始的 GAN 并没有太大的改进，将全卷积神经网络应用到了 GAN 中，细节方面，DCGAN 做了如下改进：

1. 取消 pooling 层。G 中用反卷积进行上采样，D 中用加入 stride 的卷积代替 pooling
2. batch normalization
3. 去掉 FC 层，网络为全卷积网络
4. G 中使用 Relu(最后一层用 tanh)
5. D 中用 LeakyRelu

训练细节：

1. 预处理环节，将图像 scale 到 tanh 的 [-1,1]
2. LeakyRelu 的斜率是 0.2
3. 使用 Adam 作为优化器，初始学习率 0.0002，beta1 参数设置为 0.5（处于模型稳定性考虑）

3.1 Generator

```
class NetGenerator(nn.Module):
  def __init__(self):
      super(NetGenerator,self).__init__()
      self.act1 = nn.ReLU(True)
      self.covt1 = nn.ConvTranspose2d(noiseSize, n_generator_feature * 8,
                                                  kernel_size=4, stride=1,
                                                  padding=0, bias=False)
      self.bn1 = nn.BatchNorm2d(n_generator_feature * 8)
      self.covt2 = nn.ConvTranspose2d(n_generator_feature * 8, n_generator_feature *
                                                   4, kernel_size=4, stride=2,
                                                  padding=1, bias=False)
      self.bn2 = nn.BatchNorm2d(n_generator_feature * 4)
      self.covt3 = nn.ConvTranspose2d(n_generator_feature * 4, n_generator_feature *
                                                   2, kernel_size=4, stride=2,
                                                  padding=1, bias=False)
      self.bn3 = nn.BatchNorm2d(n_generator_feature * 2)
      self.covt4 = nn.ConvTranspose2d(n_generator_feature * 2, n_generator_feature,
                                                  kernel_size=4, stride=2,
                                                  padding=1, bias=False)
      self.bn4 = nn.BatchNorm2d(n_generator_feature)
      self.covt5 = nn.ConvTranspose2d(n_generator_feature, 3, kernel_size=5, stride=
                                                  3, padding=1, bias=False)
      self.act2 = nn.Tanh()

  def forward(self, input):
      x = self.act1(self.bn1(self.covt1(input)))
      x = self.act1(self.bn2(self.covt2(x)))
      x = self.act1(self.bn3(self.covt3(x)))
      x = self.act1(self.bn4(self.covt4(x)))
      x = self.act2(self.covt5(x))
      return x
```

其中，对于转置卷积

$$H' = (H + (\text{ stride } - 1) * (H - 1) - k_h + 2 * (k_h - \text{ padding } - 1)/1 + 1$$

H 为原始长度，kh 为卷积核长度，p 为 padding 大小，s 为步长。

n_generator_feature 是全局变量在模型参数设置的生成器 feature map 数。在 Generator 的模型定义中，激活函数使用 ReLu，使用 5 层的网络，将 torch.randn 随机生成的 1*1 噪声最终转化到 96*96 大小。

在第一层将卷积核设置为 4*4，步长为 1，填充为 0，此时根据公式计算得到逆卷积之后输出的维度为 (n_generator_feature * 8) * 4 * 4；然后对输出进行归一化处理使得数据在进行 Relu 之前不会因为数据过大而导致网络性能的不稳定；再进行激活。后续 3 层中，将卷积核设置为 4*4，步长为 2，填充为 1，让每次都输出长宽维度刚刚好都是输入的 2 倍。在 2、3、4 层继续进行归一化与激活。第 5 层将卷积核设置为 5*5，步长为 3，填充为 1，需要最终输出结果，使用 tanh 将输出图片的像素映射化到 (-1,1) 空间作为返回值。

3.2 Discriminator

```
class NetDiscriminator(nn.Module):
  def __init__(self):
      super(NetDiscriminator,self).__init__()
      self.act1 = nn.LeakyReLU(0.2, inplace=True)
```

```
        self.conv1 = nn.Conv2d(3, n_discriminator_feature, kernel_size=5, stride=3,
                                                    padding=1, bias=False)
        self.conv2 = nn.Conv2d(n_discriminator_feature, n_discriminator_feature * 2,
                                                    kernel_size=4, stride=2,
                                                    padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(n_discriminator_feature * 2)
        self.conv3 = nn.Conv2d(n_discriminator_feature * 2, n_discriminator_feature *
                                                    4, kernel_size=4, stride=2,
                                                    padding=1, bias=False)
        self.bn3 = nn.BatchNorm2d(n_discriminator_feature * 4)
        self.conv4 = nn.Conv2d(n_discriminator_feature * 4, n_discriminator_feature *
                                                    8, kernel_size=4, stride=2,
                                                    padding=1, bias=False)
        self.bn4 = nn.BatchNorm2d(n_discriminator_feature * 8)
        self.conv5 = nn.Conv2d(n_discriminator_feature * 8, 1, kernel_size=4, stride=1
                                                    , padding=0, bias=False)
        self.act2 = nn.Sigmoid()

    def forward(self, input):
        x = self.act1(self.conv1(input))
        x = self.act1(self.bn2(self.conv2(x)))
        x = self.act1(self.bn3(self.conv3(x)))
        x = self.act1(self.bn4(self.conv4(x)))
        x = self.act2(self.conv5(x))
        return x.view(-1)
```

其中，对于卷积

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times p[0] - d[0] \times (kh[0] - 1)}{s[0]} + 1 \right\rfloor$$

n_discriminator_feature 是全局变量在模型参数设置的判别器 feature map 数。在 Discriminator 的模型定义中，激活函数使用 LeakyReLU。按照作者的解释，激活函数的使用无本质区别，可能用 ELU 等其他的也行，但作者说：

> 这里的选择更多的是经验总结。[1]

因此就采取和作者相同的激活函数。

在第一层将卷积核设置为 5*5，步长为 3，填充为 1，此时根据公式计算得到卷积之后输出的维度为 (n_discriminator_feature) * 32 * 32；再进行激活。后续 3 层中，将卷积核设置为 4*4，步长为 2，填充为 1，让每次都输出长宽维度刚刚好都是输入的一半。在 2、3、4 层进行归一化与激活。第 5 层卷积核设置为 4*4，步长为 1，填充为 0 需要最终输出结果，使用 Sigmoid 将输出图片的像素映射化到 (0,1) 空间作为返回值。

也就是说，Generator 生成图片和 Discriminator 判别图片的处理原理是相反对称的。

3.3 Train

将 Generator、Discriminator 和相关参数以及损失函数噪声生成集成在 GAN 的类中初始化定义，并在模型中直接集成训练和展示。下面是 GAN 中的训练函数：

```
def train(self, dataloader):
    d_every = self.d
    g_every = self.g
    for i, (image,_) in tqdm(enumerate(dataloader)):        # type((image,_)) = <class
                                                    'list'>, len((image,_)) = 2 * 256 *
                                                     3 * 96 * 96
```

```
        real_image = Variable(image)
        real_image = real_image.to('cuda')

        if (i + 1) % d_every == 0:
            self.optimizer_d.zero_grad()
            output = self.Discriminator(real_image)      # 尽可能把真图片判为True
            error_d_real = self.criterion(output, self.true_labels)
            error_d_real.backward()

            self.noises.data.copy_(torch.randn(self.batch_size, self.noiseSize, 1, 1))
            fake_img = self.Generator(self.noises).detach()       # 根据噪声生成假图
            fake_output = self.Discriminator(fake_img)       # 尽可能把假图片判为False
            error_d_fake = self.criterion(fake_output, self.fake_labels)
            error_d_fake.backward()
            self.optimizer_d.step()

        if (i + 1) % g_every == 0:
            self.optimizer_g.zero_grad()
            self.noises.data.copy_(torch.randn(self.batch_size, self.noiseSize, 1, 1))
            fake_img = self.Generator(self.noises)        # 这里没有detach
            fake_output = self.Discriminator(fake_img)       # 尽可能让Discriminator把
                                                               假图片判为True
            error_g = self.criterion(fake_output, self.true_labels)
            error_g.backward()
            self.optimizer_g.step()
            ## 在训练判别器时，需要对生成器生成的图片用detach操作进行计算图截断，避免
                                                               反向传播将梯度传到生成器
                                                               中。因为在训练判别器时我们
                                                               不需要训练生成器，也就不需
                                                               要生成器的梯度。
```

其中，d_every 定义每几个 batch 训练一次 discriminator；g_every 定义每几个 batch 训练一次 generator。

训练时，对于假图片，在训练判别器时，尽量输出 0；训练生成器时，尽量输出 1。训练生成器时，无须调整判别器的参数；训练判别器时，无须调整生成器的参数。

3.4 结果

将 batch size 设置为 512 总共训练了 66 次迭代（因为在之前的测试中 79 次的时候内存爆了），修改模型都过程中的一次训练录频见：

myx 的录频 video

设置 plot_epoch = [1,5,10,20,30,40,50,60,65,66]，在这几个结果进行输出生成的动漫头像如下：

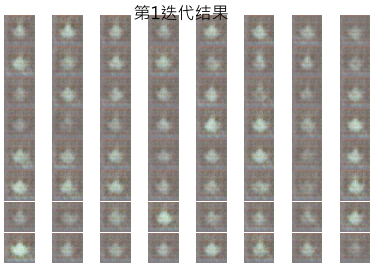

图 3: 第 1 次迭代

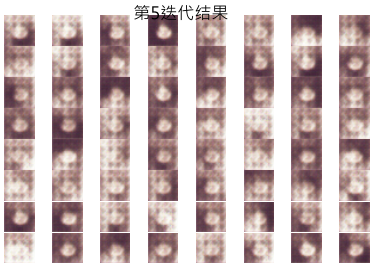

图 4: 第 5 次迭代

图 5: 第 10 次迭代

图 6: 第 20 次迭代

图 7: 第 30 次迭代

图 8: 第 40 次迭代

图 9: 第 50 次迭代

图 10: 第 60 次迭代

图 11: 第 65 次迭代

图 12: 第 66 次迭代

看得出训练的效果还是很好的，从噪声数据很快地在迭代中输出了肉眼可见是动漫头像的图像。

参考文献

[1] SICHENG Y. 生成对抗网络实战——用 GAN 生成动漫头像[EB/OL]. 2021. https://blog.csdn.net/qq_41897800/article/details/115639025.