

人工智能作业3

1852824 吴杨婉婷

- [人工智能作业3](#)
 - [1. 作业需求描述](#)
 - [2. Requirements](#)
 - [3. 模型代码详解](#)
 - [3.1 原始gan生成器](#)
 - [3.2 原始gan鉴别器](#)
 - [3.3 DCgan生成器](#)
 - [3.4 DCgan鉴别器](#)
 - [4.实验结果](#)
 - [4.1 原始gan](#)
 - [4.2 dcgan](#)
 - [5.结果分析与心得体会](#)
 - [5.1理论GAN](#)
 - [5.2 GAN的优点](#)
 - [5.3 GAN的缺点](#)
 - [5.4 DCGAN网络结构设计要点](#)
 - [5.5 DCGAN训练细节](#)
 - [5.6 总结](#)
 - [6.主体代码解释](#)
 - [7.作者](#)

1. 作业需求描述

使用原始GAN，DCGAN实现手写数字生成

2. Requirements

- **Development Environment:**

Win 10

- **Development Software:**

PyCharm 2020.3.5.PC-191.6605.12

- **Development Language:**

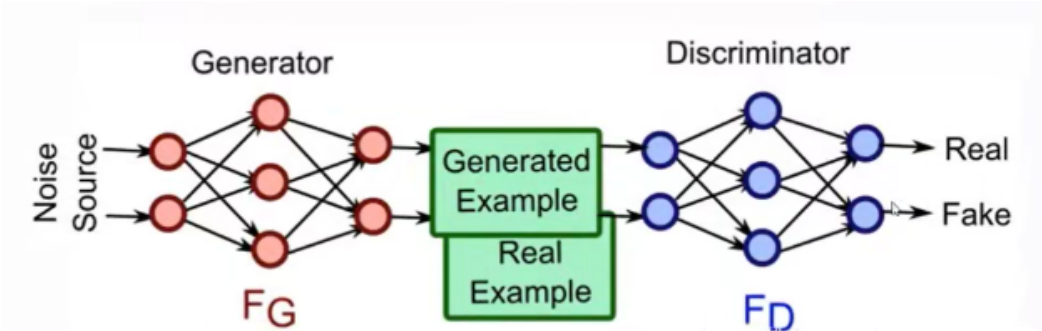
Python

- **Mainly Reference Count:**

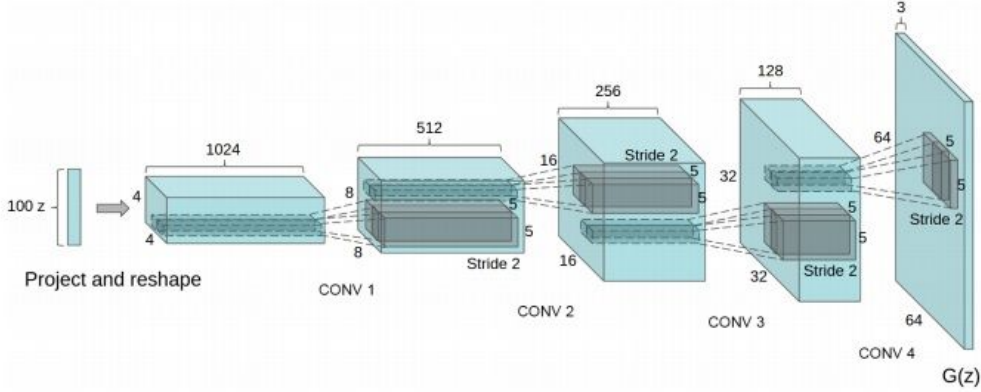
1. torchvision
2. matplotlib
3. os

- 4. torch
- 5. numpy

3. 模型代码详解

number	描述	model
1	<p>生成器有三个全链接隐藏层，每层的参数分别是32，64，128，每层加上leaky_relu激活函数和dropout，判别器是生成器的逆转</p> 	原始GAN

DCGAN主要是在网络架构上改进了原始GAN，DCGAN的生成器与判别器都利用CNN架构替换了原始GAN的全连接网络，主要改进之处有如下几个方面：DCGAN的生成器和判别器都舍弃了CNN的池化层，判别器保留CNN的整体架构，生成器则是将卷积层替换成了反卷积层。在判别器和生成器中在每一层之后都是用了BN层，有助于处理初始化不良导致的训练问题，加速模型训练，提升了训练的稳定性。利用1*1卷积层替换到所有的全连接层。在生成器中除输出层使用Tanh激活函数，其余层全部使用ReLU激活函数。在判别器所有层都使用LeakyReLU激活函数，防止梯度稀疏。

2		DCGAN
---	--	-------

3.1 原始gan生成器

```
class Generator(nn.Module):  
  
    def __init__(self, input_size=100, hidden=(32, 64, 128), output_size=784):  
        super(Generator, self).__init__()  
  
        # 隐藏层  
        self.fc1 = nn.Linear(input_size, hidden[0])  
        self.fc2 = nn.Linear(hidden[0], hidden[1])  
        self.fc3 = nn.Linear(hidden[1], hidden[2])
```

```

# 输出层
self.fc4 = nn.Linear(hidden[2], output_size)

# Dropout
self.dropout = nn.Dropout(.3)

def forward(self, x):
    x = F.leaky_relu(self.fc1(x))
    x = self.dropout(x)
    x = F.leaky_relu(self.fc2(x))
    x = self.dropout(x)
    x = F.leaky_relu(self.fc3(x))
    x = self.dropout(x)
    out = torch.tanh(self.fc4(x))
    return out

```

3.2 原始gan鉴别器

```

class Discriminator(nn.Module):

    def __init__(self, input_size=784, hidden=(128, 64, 32), output_size=1):
        super(Discriminator, self).__init__()

        # 隐藏层
        self.fc1 = nn.Linear(input_size, hidden[0])
        self.fc2 = nn.Linear(hidden[0], hidden[1])
        self.fc3 = nn.Linear(hidden[1], hidden[2])

        # 输出层
        self.fc4 = nn.Linear(hidden[2], output_size)

        # Dropout
        self.dropout = nn.Dropout(.3)

    def forward(self, x):
        x = x.view(-1, 784)
        x = F.leaky_relu(self.fc1(x))
        x = self.dropout(x)
        x = F.leaky_relu(self.fc2(x))
        x = self.dropout(x)
        x = F.leaky_relu(self.fc3(x))
        x = self.dropout(x)
        out = self.fc4(x)
        return out

```

3.3 DCgan生成器

```

class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        self.init_size = opt.img_size // 4
        self.l1 = nn.Sequential(nn.Linear(opt.latent_dim, 128 * self.init_size **
2))

        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, opt.channels, 3, stride=1, padding=1),
            nn.Tanh(),
        )
    def forward(self, z):
        out = self.l1(z)
        out = out.view(out.shape[0], 128, self.init_size, self.init_size)
        img = self.conv_blocks(out)
        return img

```

3.4 DCgan鉴别器

```

class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        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), nn.Dropout2d(0.25)]
            if bn:
                block.append(nn.BatchNorm2d(out_filters, 0.8))
            return block
        self.model = nn.Sequential(
            *discriminator_block(opt.channels, 16, bn=False),
            *discriminator_block(16, 32),
            *discriminator_block(32, 64),
            *discriminator_block(64, 128),
        )
        ds_size = opt.img_size // 2 ** 4
        self.adv_layer = nn.Sequential(nn.Linear(128 * ds_size ** 2, 1),
nn.Sigmoid())
    def forward(self, img):
        out = self.model(img)
        out = out.view(out.shape[0], -1)

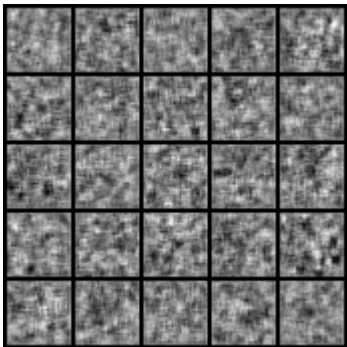



```






```
validity = self.adv_layer(out)
return validity
```

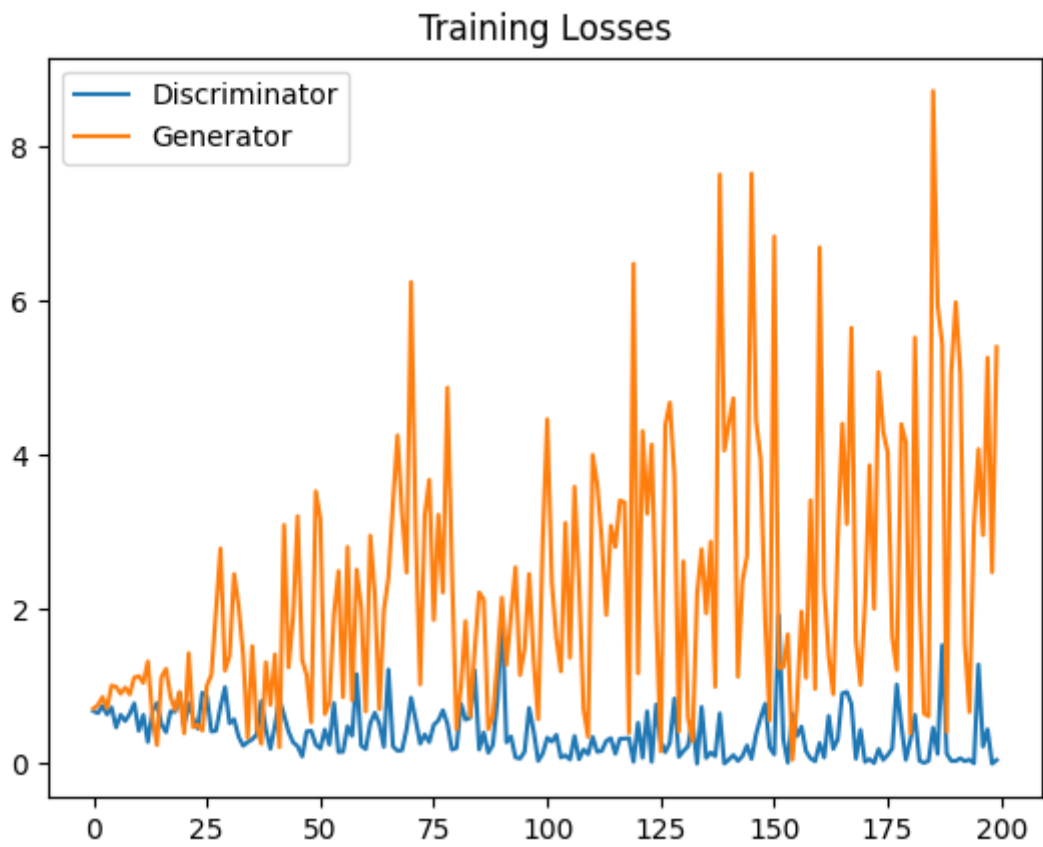
4.实验结果

4.1 原始gan

可以注意到原始gan在训练量比较小的时候效果非常不好，生成的图片非常模糊，把epoch增加到200生产器的效果才变好

number	pic
0	
200	
400	
2000	






number	pic
5400	
96000	
124000	
152000	
184000	






•

4.2 dcgan

batch_number	pic
0	
4000	

batch_number	pic
8000	
12000	
40000	
68000	
96000	

batch_number	pic
124000	
152000	
184000	

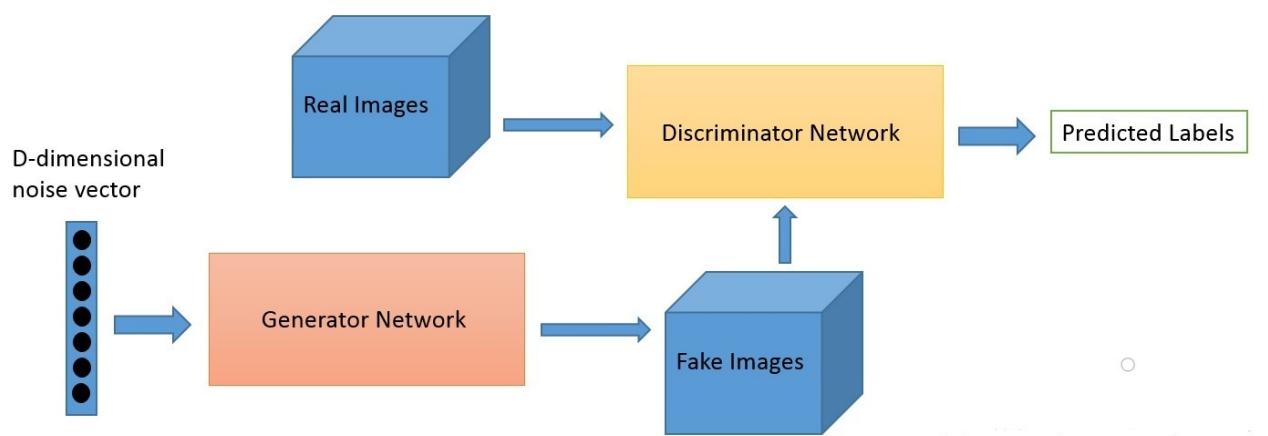


5.结果分析与心得体会

5.1理论GAN

- GAN主要包括了两个部分，即生成器generator与判别器discriminator。生成器主要用来学习真实图像分布从而让自身生成的图像更加真实，以骗过判别器。判别器则需要对接收的图片进行真假判别。在整个过程中，生成器努力地让生成的图像更加真实，而判别器则努力地去识别出图像的真假，这个过程相当于一个二人博弈，随着时间的推移，生成器和判别器在不断地进行对抗，最终两个网络达到了一个动态均衡：生成器生成的图像接近于真实图像分布，而判别器识别不出真假图像，对于给定图像的预测为真的概率基本接近0.5（相当于随机猜测类别）。
- 对于GAN更加直观的理解可以用一个例子来说明：造假币的团伙相当于生成器，他们想通过伪造金钱来骗过银行，使得假币能够正常交易，而银行相当于判别器，需要判断进来的钱是真钱还是假币。因此假币团伙的目的是要造出银行识别不出的假币而骗过银行，银行则是要想办法准确地识别出假币。
- 因此，我们可以将上面的内容进行一个总结。给定真=1，假=0，那么有：对于给定的真实图片，判别器要为其打上标签1；对于给定的生成图片，判别器要为其打上标签0；对于生成器传给判别器的生成图

片，生成器希望判别器打上标签1。



5.2 GAN的优点

1. GAN采用的是一种无监督的学习方式训练，可以被广泛应用在无监督学习和半监督学习领域；
2. GAN应用到一些场景上，比如图像风格迁移，超分辨率，图像补全，去噪，避免了损失函数设计的困难，只要有一个的基准，直接加上判别器，剩下的就交给对抗训练了。

5.3 GAN的缺点

1. 训练GAN需要达到纳什均衡,有时候可以用梯度下降法做到,有时候做不到
2. GAN不适合处理离散形式的数据,比如文本
3. GAN存在训练不稳定、梯度消失、模式崩溃/坍塌的问题
4. 难以训练,需要小心设置网络训练中的参数。

5.4 DCGAN网络结构设计要点

1. 在D网络中用strided 卷积代替pooling层，在G网络中用ConvTranspose2d代替上采样层。
2. 在G和D网络中直接将BN应用到所有层会导致样本震荡和模型不稳定，通过在generator输出层和discriminator输入层不采用BN可以防止这种现象。但实际情况还是会有样本震荡
3. G网络中除了输出层tanh都使用ReLU激活函数
4. D网络中都使用LeakyReLU激活函数

5.5 DCGAN训练细节

1. 预处理环节，将图像scale到tanh的[-1, 1]。
2. mini-batch训练，batch size是128.
3. 所有的参数初始化由(0, 0.02)的正态分布中随即得到
4. LeakyReLU的斜率是0.2.
5. 虽然之前的GAN使用momentum来加速训练，DCGAN使用调好超参的Adam optimizer。
6. learning rate=0.0002
7. 将momentum参数beta从0.9降为0.5来防止震荡和不稳定。

5.6 总结

1. 能用Adam优化器的情况下尽量使用Adam优化器，不行的话使用RMSprop优化器。
2. DCGAN训练过程中，通常是训练判别器，然后训练整个DCGAN。并且DCGAN训练中，判别器不能更新参数，因此必须冻结所有层。为了加快网络训练，通常是训练多次判别器，然后训练一次生成器，这样

能提高网络训练速率。(但是实际效果不是很明显, 可能是mnist数据集比较小)

3. 从结果上可以清楚发现原始gan的生成效果一般, epoch增加到50以上才可以看清楚大致的数字, 而dcgan从第二次迭代就可以出现大致的形状了, 复杂网络可能会有更好的效果。

6.主体代码解释

以dcgan为例

- 方便手动调整参数

```
os.makedirs("images", exist_ok=True)
parser = argparse.ArgumentParser()
parser.add_argument("--n_epochs", type=int, default=200, help="number of epochs of training")
parser.add_argument("--batch_size", type=int, default=64, help="size of the batches")
parser.add_argument("--lr", type=float, default=0.0002, help="adam: learning rate")
parser.add_argument("--b1", type=float, default=0.5, help="adam: decay of first order momentum of gradient")
parser.add_argument("--b2", type=float, default=0.999, help="adam: decay of first order momentum of gradient")
parser.add_argument("--n_cpu", type=int, default=8, help="number of cpu threads to use during batch generation")
parser.add_argument("--latent_dim", type=int, default=100, help="dimensionality of the latent space")
parser.add_argument("--img_size", type=int, default=32, help="size of each image dimension")
parser.add_argument("--channels", type=int, default=1, help="number of image channels")
parser.add_argument("--sample_interval", type=int, default=4000, help="interval between image sampling")
opt = parser.parse_args()
print(opt)
```

- 权重初始化函数

```
cuda = True if torch.cuda.is_available() else False

def weights_init_normal(m): #权重初始化
    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)
```

- 生成器

```

class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        self.init_size = opt.img_size // 4
        self.l1 = nn.Sequential(nn.Linear(opt.latent_dim, 128 * self.init_size **
2))

        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, opt.channels, 3, stride=1, padding=1),
            nn.Tanh(),
        )
    def forward(self, z):
        out = self.l1(z)
        out = out.view(out.shape[0], 128, self.init_size, self.init_size)
        img = self.conv_blocks(out)
        return img

```

- 鉴别器

```

class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        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), nn.Dropout2d(0.25)]
            if bn:
                block.append(nn.BatchNorm2d(out_filters, 0.8))
            return block
        self.model = nn.Sequential(
            *discriminator_block(opt.channels, 16, bn=False),
            *discriminator_block(16, 32),
            *discriminator_block(32, 64),
            *discriminator_block(64, 128),
        )
        ds_size = opt.img_size // 2 ** 4
        self.adv_layer = nn.Sequential(nn.Linear(128 * ds_size ** 2, 1),
nn.Sigmoid())
    def forward(self, img):
        out = self.model(img)
        out = out.view(out.shape[0], -1)
        validity = self.adv_layer(out)

```

```
        return validity
adversarial_loss = torch.nn.BCELoss()
```

- 初始化生成器, 鉴别器

```
generator = Generator()
discriminator = Discriminator()

if cuda:
    generator.cuda()
    discriminator.cuda()
    adversarial_loss.cuda()
```

- 初始化权重

```
generator.apply(weights_init_normal)
discriminator.apply(weights_init_normal)
```

- 导入数据

```
os.makedirs("./data/mnist", exist_ok=True)
dataloader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "./data",
        train=True,
        download=True,
        transform=transforms.Compose(
            [transforms.Resize(opt.img_size), transforms.ToTensor(),
             transforms.Normalize([0.5], [0.5])]
        ),
        batch_size=opt.batch_size,
        shuffle=True,
    )
)
```

- 优化器设置

```
optimizer_G = torch.optim.Adam(generator.parameters(), lr=opt.lr, betas=(opt.b1,
opt.b2))
optimizer_D = torch.optim.Adam(discriminator.parameters(), lr=opt.lr, betas=
(opt.b1, opt.b2))
```

```
Tensor = torch.cuda.FloatTensor if cuda else torch.FloatTensor
```

- 开始训练

```
all_loss=[]
all_loss2=[]
for epoch in range(opt.n_epochs):
    for i, (imgs, _) in enumerate(dataloader):

        valid = Variable(Tensor(imgs.shape[0], 1).fill_(1.0), requires_grad=False)
        fake = Variable(Tensor(imgs.shape[0], 1).fill_(0.0), requires_grad=False)

        real_imgs = Variable(imgs.type(Tensor))

        # 生成器开始训练

        optimizer_G.zero_grad()

        # 简单噪声
        z = Tensor(np.random.normal(0, 1, (imgs.shape[0], opt.latent_dim)))
        gen_imgs = generator(z)

        g_loss = adversarial_loss(discriminator(gen_imgs), valid)

        g_loss.backward()
        optimizer_G.step()

        # 鉴别器开始训练

        optimizer_D.zero_grad()

        real_loss = adversarial_loss(discriminator(real_imgs), valid)
        fake_loss = adversarial_loss(discriminator(gen_imgs.detach()), fake)
        d_loss = (real_loss + fake_loss) / 2

        d_loss.backward()
        optimizer_D.step()

        batches_done = epoch * len(dataloader) + i
        if batches_done % opt.sample_interval == 0:
            save_image(gen_imgs.data[:25], "images/%d.png" % batches_done, nrow=5,
normalize=True)
            print(
                "[Epoch %d/%d] [Batch %d/%d] [D loss: %f] [G loss: %f]"
                % (epoch, opt.n_epochs, i, len(dataloader), d_loss.item(), g_loss.item())
            )
            all_loss.append(d_loss.item())
            all_loss2.append(g_loss.item())
```

- 保存参数信息

```
PATH1= 'generator.pkl'  
PATH2= 'discriminator.pkl'  
torch.save(generator,PATH1)  
torch.save(discriminator,PATH2)
```

- 作图

```
plt.plot(all_loss, label='Discriminator')  
plt.plot(all_loss2, label='Generator')  
plt.title("Training Losses")  
plt.legend()  
plt.savefig("dcgan")
```

7.作者

ID	Name
1852824	吴杨婉婷

指导老师 唐堂老师

联系方式 email: 1852824@tongji.edu.cn