

姓名：杨航 学号：1811451

Conditional-GAN by Jittor

环境配置

计图要求的环境是：

- 操作系统: **Linux**(e.g. Ubuntu/CentOS/Arch) 或 **Windows Subsystem of Linux (WSL)**
- Python: 版本 ≥ 3.7
- C++编译器 (需要下列至少一个)
 - g++ ($\geq 5.4.0$)
 - clang (≥ 8.0)

我的环境：

- 操作系统: **Linux Ubuntu20**
- Python: 3.8.0 64bit
- C++编译器
 - g++ 7.5.0

网络搭建--基础网络

(基础网络是按照计图官方教程的思路来搭建的)

- 需要引入的库：

```
# jittor 算子
import jittor as jt
# jittor 的初始化和随机函数
from jittor import init, rand, randint
# 系统库，用于创建文件夹等操作
import os
import numpy as np
# jittor的neuronnetwork操作库
from jittor import nn
# 我自己创建的数据集类
from mydataset import myDataset
# 图片处理
from PIL import Image
# jittor自带mnist库
from jittor.dataset.mnist import MNIST
# 用于做一些预处理
import jittor.transform as transform
```

- Generator

思路解释写在下面代码的注释上了：

```

class Generator(nn.Module):
    """
    全连接generator:
    """
    def __init__(self):
        # use nn.Module's __init__() to initialize generator. super means
        # generator is a superclass of
        super(Generator, self).__init__()
        # generate data from 1~10, and each for 10, so it's 10*10
        self.label_emb = nn.Embedding(opt.n_classes, opt.n_classes)
        # use block to define layers, used in nn.Sequential to generate
        # sequential layers
        def block(in_feat, out_feat, normalize=True):
            """
            each block:
            1. linner layer (in_feat->out_feat)
            2. if normalize==true, add batchnorm1d layer
            3. leakyrelu layer
            """
            layers = [nn.Linear(in_feat, out_feat)]
            if normalize: # BatchNorm就是在深度神经网络训练过程中使得每一层神经网络
                # 的输入保持相同分布的。
                layers.append(nn.BatchNorm1d(out_feat, 0.8))
            layers.append(nn.LeakyReLU(0.2))
            return layers
        # Sequential would add activated funtion automatically
        self.model = nn.Sequential(*block((opt.latent_dim + opt.n_classes),
        128, normalize=False),
        *block(128, 256),
        *block( 256, 512),
        *block(512, 1024),
        nn.Linear(1024, int(np.prod(img_shape))),
        # generate a picture
        nn.Tanh())

    def execute(self, noise, labels):
        gen_input = jt.contrib.concat((self.label_emb(labels), noise),
        dim=1)
        img = self.model(gen_input)
        img = img.view((img.shape[0], *img_shape))
        return img

```

- Discriminator

```

class Discriminator(nn.Module):
    """
    discriminator 的思路和generator差不多
    """
    def __init__(self):
        # 作为超类先初始化
        super(Discriminator, self).__init__()
        # 同上讲类标签向量化
        self.label_embedding = nn.Embedding(opt.n_classes, opt.n_classes)
        # 这里也都是全连接层

```

```

        self.model = nn.Sequential(nn.Linear((opt.n_classes +
int(np.prod(img_shape))), 512),

                                nn.LeakyReLU(0.2),
                                nn.Linear(512, 512),
                                nn.Dropout(0.4),
                                nn.LeakyReLU(0.2),
                                nn.Linear(512, 256),
                                nn.Dropout(0.4),
                                nn.LeakyReLU(0.2),
                                nn.Linear(256, 1))

    def execute(self, img, labels):
        d_in = jt.contrib.concat((img.view((img.shape[0], (- 1))),
self.label_embedding(labels)), dim=1)
        validity = self.model(d_in)
        return validity

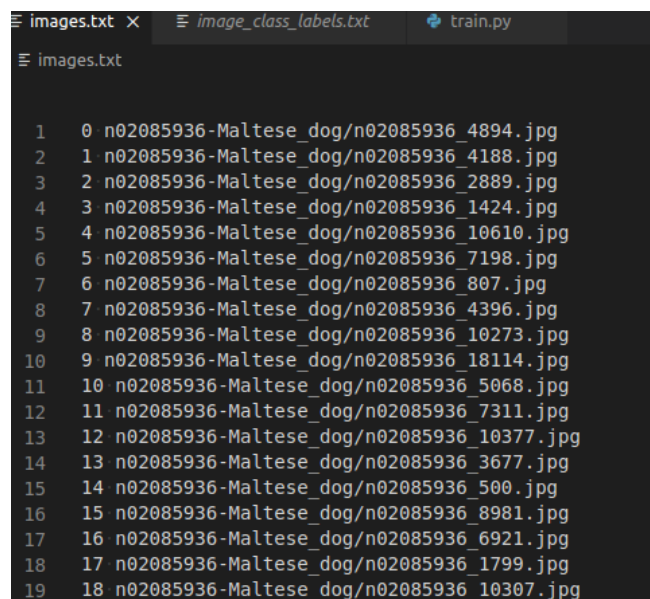
```

- myDataset类

1. 先生成数据集中图片的路径:

我的数据集文件组织形式为: 每个类是一个文件夹, 每个文件夹下的图片都属于这个类。

- 首先先为每个图片生成索引, 并保存其相对路径;



```

images.txt
1 0 n02085936-Maltese_dog/n02085936_4894.jpg
2 1 n02085936-Maltese_dog/n02085936_4188.jpg
3 2 n02085936-Maltese_dog/n02085936_2889.jpg
4 3 n02085936-Maltese_dog/n02085936_1424.jpg
5 4 n02085936-Maltese_dog/n02085936_10610.jpg
6 5 n02085936-Maltese_dog/n02085936_7198.jpg
7 6 n02085936-Maltese_dog/n02085936_807.jpg
8 7 n02085936-Maltese_dog/n02085936_4396.jpg
9 8 n02085936-Maltese_dog/n02085936_10273.jpg
10 9 n02085936-Maltese_dog/n02085936_18114.jpg
11 10 n02085936-Maltese_dog/n02085936_5068.jpg
12 11 n02085936-Maltese_dog/n02085936_7311.jpg
13 12 n02085936-Maltese_dog/n02085936_10377.jpg
14 13 n02085936-Maltese_dog/n02085936_3677.jpg
15 14 n02085936-Maltese_dog/n02085936_500.jpg
16 15 n02085936-Maltese_dog/n02085936_8981.jpg
17 16 n02085936-Maltese_dog/n02085936_6921.jpg
18 17 n02085936-Maltese_dog/n02085936_1799.jpg
19 18 n02085936-Maltese_dog/n02085936_10307.jpg

```

- 然后为每个索引的图片生成其label

```
image_class_labels.txt
1 0 0
2 1 0
3 2 0
4 3 0
5 4 0
6 5 0
7 6 0
8 7 0
9 8 0
10 9 0
11 10 0
12 11 0
13 12 0
14 13 0
15 14 0
16 15 0
17 16 0
18 17 0
19 18 0
```

- 最后生成划分train和test的文件（1是train，0是test）

```
train_test_split.txt
1 0 1
2 1 1
3 2 1
4 3 1
5 4 1
6 5 1
7 6 1
8 7 1
9 8 1
10 9 1
11 10 1
12 11 1
13 12 1
14 13 1
15 14 1
16 15 1
17 16 1
18 17 1
19 18 1
```

2. mydataset导入数据集信息：

```
class myDataset(Dataset):
    """
    # Description:
        Dataset for retrieving CUB-200-2011 images and labels

    # Member Functions:
        __init__(self, phase, resize): initializes a dataset
            phase: a string in ['train', 'val',
'test']
            resize: output shape/size of an image

        __getitem__(self, item): returns an image
            item: the index of image in the whole
dataset

        __len__(self): returns the length of dataset
    """

    def __init__(self, phase='train', resize=(300,300)):
```

```

        super().__init__() # necessary, without this line, not succeeded
from jittor.dataset.dataset Dataset
    assert phase in ['train', 'val', 'test']
    self.phase = phase
    self.resize = resize
    self.image_id = []
    self.num_classes = 2

    # get image path from images.txt
    with open(os.path.join(DATAPATH, 'images.txt')) as f:
        for line in f.readlines():
            line = line.strip()
            i_space = line.index(' ')
            id = line[:i_space]
            path = line[i_space+1:]
            #id, path = line.strip().split(' ')
            image_path[id] = path

    # get image label from image_class_labels.txt
    with open(os.path.join(DATAPATH, 'image_class_labels.txt')) as
f:
        for line in f.readlines():
            id, label = line.strip().split(' ')
            image_label[id] = int(label)

    # get train/test image id from train_test_split.txt
    with open(os.path.join(DATAPATH, 'train_test_split.txt')) as f:
        for line in f.readlines():
            image_id, is_training_image = line.strip().split(' ')
            is_training_image = int(is_training_image)

            if self.phase == 'train' and is_training_image:
                self.image_id.append(image_id)
            if self.phase in ('val', 'test') and not
is_training_image:
                self.image_id.append(image_id)

    # transform
    self.transform = transform

    def __getitem__(self, item):
        # get image id
        image_id = self.image_id[item]

        # image
        image = Image.open(os.path.join(Image_Dir, 'yb',
image_path[image_id])).convert('RGB') # (C, H, W)
        image = self.transform(image)

        # return image and label
        return image, image_label[image_id] # count begin from zero

    def __len__(self):
        return len(self.image_id)

```

3. dataloader真正导入图片:

```
dogData=myDataset()  
dataloader = dogData.set_attrs(batch_size=opt.batch_size, shuffle=True)
```

网络训练

- 训练generator

输入是一个100维度的隐空间（增加生成结果多样性），和控制条件（限定生成图片的label）

返回生成图片之后，将生成的假图片输入到discriminator中计算loss的大小来指导generator网络权值更新

其中 `g_loss.sync()` 函数因为查文档文档也写不清楚，我就去github上提了issue，官方回复很快，说这个是强制同步的意思，如果没有它就是懒同步，所以删除而已没有关系。

```
# Sample noise and labels as generator input  
z = jt.array(np.random.normal(0, 1, (batch_size,  
opt.latent_dim))).float32()  
gen_labels = jt.array(np.random.randint(0, opt.n_classes,  
batch_size)).float32()  
  
# Generate a batch of images  
gen_imgs = generator(z, gen_labels)  
# Loss measures generator's ability to fool the discriminator  
validity = discriminator(gen_imgs, gen_labels)  
g_loss = adversarial_loss(validity, valid)  
g_loss.sync()  
optimizer_G.step(g_loss)
```

- 训练discriminator

鉴别器类似一个分类器，但是它不具体分类，它只鉴别图片真伪。

训练的步骤如下：

- 先给定真的图片，并加入其正确标签，计算真图判别loss
- 在给定生成器生成的图片机器标签，计算假图判别loss
- 最后用两个loss的加权（各0.5）作为其更新优化的指导

```
# Loss for real images  
validity_real = discriminator(real_imgs, labels)  
d_real_loss = adversarial_loss(validity_real, valid)  
  
# Loss for fake images  
validity_fake = discriminator(gen_imgs.stop_grad(), gen_labels)  
d_fake_loss = adversarial_loss(validity_fake, fake)  
  
# Total discriminator loss  
d_loss = (d_real_loss + d_fake_loss) / 2  
d_loss.sync()  
optimizer_D.step(d_loss)
```

- 保存模型和学习率递减

每5个epoch保存一次模型，以及没10个epoch学习率减小到原来的0.9

```

sample_generate( batches_done=batches_done)
if epoch % 5 == 0:
    generator.save("saved_models/generator_last.pkl")
    discriminator.save("saved_models/discriminator_last.pkl")
if epoch % 10 == 0:
    opt.lr*=0.9

```

卷积层引进generator

为什么generator引入卷积

由于官方教程中的模型都是使用的全连接层，全连接层有以下问题

- 参数太多，模型过大，我的pc内存装不下。这样的话我就得迫使输出的图片尺寸减小
- 全连接层的学习都是线性的，相比于卷积的方式，可能对图片的方向，位置，大小等信息敏感

```

class Generator(nn.Module):

    def __init__(self):
        super(Generator, self).__init__()
        self.label_emb = nn.Embedding(opt.n_classes, 10)
        input_dim = ((opt.latent_dim + 10) )
        self.init_size = (opt.img_size // 4)
        self.ll = nn.Sequential(nn.Linear(input_dim, (128 * (self.init_size
** 2))))
        self.conv_blocks = nn.Sequential(nn.BatchNorm(128),
                                         nn.Upsample(scale_factor=2),
                                         nn.Conv(128, 128, 3, stride=1,
padding=1),
                                         nn.BatchNorm(128, eps=0.8),
                                         nn.LeakyReLU(scale=0.2),
                                         nn.Upsample(scale_factor=2),
                                         nn.Conv(128, 64, 3, stride=1,
padding=1),
                                         nn.BatchNorm(64, eps=0.8),
                                         nn.LeakyReLU(scale=0.2),
                                         nn.Conv(64, opt.channels, 3,
stride=1, padding=1),
                                         nn.Tanh())

        def weights_init_normal(m):
            classname = m.__class__.__name__
            if (classname.find('Conv') != (- 1)):
                init.gauss_(m.weight, mean=0.0, std=0.02)
            elif (classname.find('BatchNorm') != (- 1)):
                init.gauss_(m.weight, mean=1.0, std=0.02)
                init.constant_(m.bias, value=0.0)
            for m in self.modules():
                weights_init_normal(m)

        def execute(self, noise, labels):
            gen_input = jt.contrib.concat((self.label_emb(labels), noise),
dim=1)
            out = self.ll(gen_input)
            out = out.view((out.shape[0], 128, self.init_size, self.init_size))
            img = self.conv_blocks(out)
            return img

```

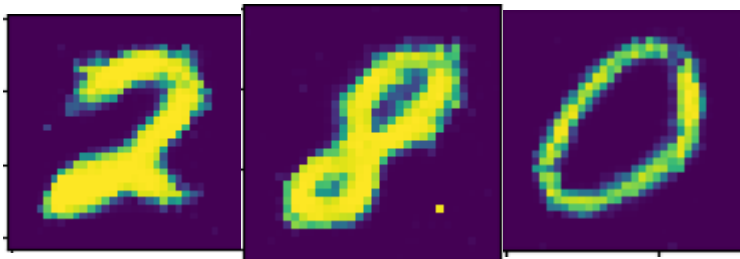
为什么鉴别器不用卷积

因为在鉴别器中，输入的是图片以及把标签向量化之后concat在一起。如果用卷积自动学习图片的特征的话，那向量化后的label的部分其实本不是图片的特征，但是会被卷积层卷积成为其特征，这样不太合理。所以我最后还是把决定在鉴别器中使用全连接层。

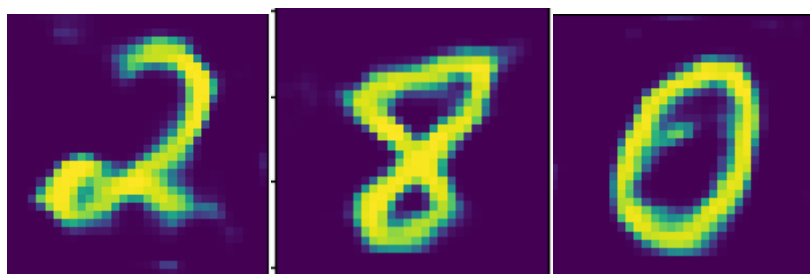
实验结果

手写数字数据集

- 全连接方式的生成器结果：



- 卷积方式的生成器



我们可以看出在手写数据集上，两种方法的效果差不多但是呈现以下差异

- 全连接的方式的手写数字边缘不清晰，会有严重的雾化感，因此会导致看起来不太真实
- 在全连接的“8”中，右下角有一个黄色的方框，这也是它会出现的artifact之一
- 卷积方式中，边缘相对更加平滑，看起来更像是笔写出来的字迹。
- 但是在卷积方式中，背景本该没有字迹的地方也会出现一些非背景颜色，以及上述卷积方式生成的“0”上有一块多余的区域，这是它会出现的artifact
- 卷积方式的训练中，大概20个epoch就能得到较好的效果，但是线性模型可能需要到50的时候才能10个数字都取得较好的效果。

我自己的数据集

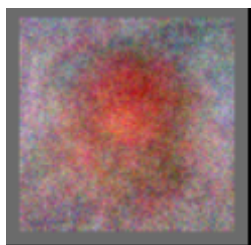
除了手写字迹数据集之外，我还自己找了两个数据集，一个是狗数据集，一个是花卉小数据集，发现两个数据集的表现都非常不好

- 狗的全连接方式：（总20700个图片）



其实可以隐约看出是一直狗的侧面，但是不太清楚

- 花卉的全连接方式：（1360个图片）



非常模糊，颗粒感非常强，且感觉像是把花卉的各个面结合在一起了，因为数据集中既有花的侧麦呢，又有花的正面

- 花卉的卷积方式：



分块感较强，我认为这里的效果不好是因为，最后我的网络只能承受80*80像素大小的图片，因此首先图片被下采样到很模糊的状态，而卷积核最小是3*3的，因此很多细节都被大粒度的滤掉了。最后就呈现了上图的比较块状不清晰效果。

训练集测试集结果

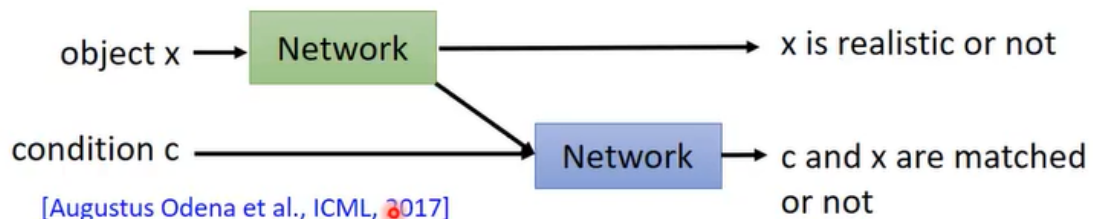
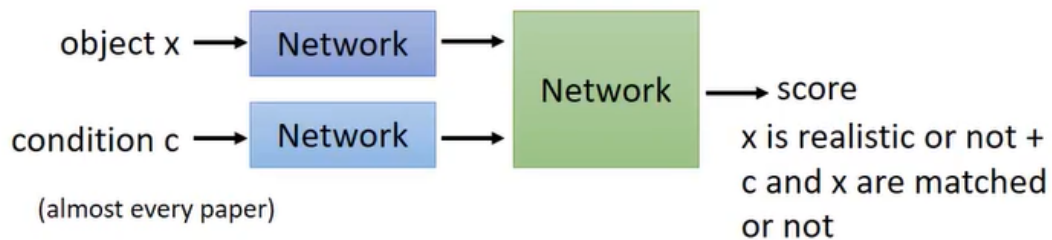
因为手写数字数据集效果比较好，我们再这上面测试，测定训练集和测试集在最后的模型上的平均loss值：

- 训练集真图平均MSE loss：11.46 生成图：5.4
- 测试集真图平均MSE loss：5.35 生成图：1.85

所以在我们的模型上，不论是训练集还是测试集，都是生成的假图鉴别效果比较好。我们的模型表现出的泛化能力比较好，因为其在测试集上的真图测试loss比较低，而在训练集上反而搞出两倍多。这个可能是由于训练的时候，真图loss只占了最终知道鉴别器的对抗loss的1/2的话语权，所以它在本数据集上看到真图并判定为真图的时候，有可能因为生成器的loss太高而导致最后对抗loss比较高，而导致认为自己的判断是错误的，因此在训练集的真图鉴别效果反而不好。

结论

1. Cgan在思路解决了如何控制gan的输出问题
2. cgan目前只能在比较简单的数据上得到非常好的效果，例如手写数据集，生成词向量
3. 猜想，或许在好的显卡上以及拥有更大的数据集之后，效果会在复杂数据集上变好
 - 更好的显卡允许原始训练图片不用下采样到80* 80这么小的尺寸，很多细节特征得以保留
 - 更大的数据集能提供更多信息供模型学习
4. 改进数据集的分布或许可以提高生成效果
 - 因为我的狗（或者花）数据集都是有各种角度各种背景的，不够细粒度，导致生成的图片依然可能既像正面又像侧面
5. 改进网络结构或许可以提高生成效果



[Augustus Odena et al., ICML, 2017]

[Takeru Miyato, et al., ICLR, 2018]

[Han Zhang, et al., arXiv, 2017]

在查资料的时候，发现discriminator有以上两种结构，

上面的结构是我的方法用的结构

下面的结构是另一种方式，其主要思想就是：训练两个discrimnaor，第一个告诉我们生成的图片是否逼真，然后第二个discriminator获得第一个d的结果和condition结合作为输入，然后输出是否这个生成的图片和我们给的条件相吻合，这样的思路似乎要更加合理

因此改变网路结构也或许能让结果更好。