西安交通大学
XI'AN JIAOTONG UNIVERSITY

人工智能技术导论

电气工程学院

生成模型：GAN

GAN

生成式对抗网络(Generative Adversarial Networks，GAN)

是一种生成式机器学习模型，是近年来复杂分布上无监督学习最具前景的方法之一。

最初，GAN由Ian J. Goodfellow于2014年发明，并在论文Generative Adversarial Nets[1]中首次进行了描述，其主要由两个不同的模型共同组成——生成器(Enerative Model)和判别器(Discriminative Model)：

- 生成器的任务是生成看起来像训练图像的**假图像**；
- 判别器需要判断从生成器输出的图像是**真实的训练图像**还是**假图像**。

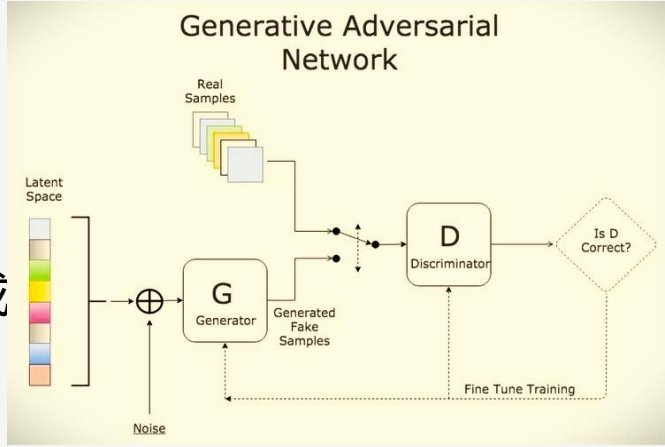
GAN模型的核心在于提出了通过对抗过程来估计生成模型这一全新框架。在这个框架中，将会同时训练两个模型——捕捉数据分布的生成模型 ◆和估计样本是否来自训练数据的判别模型 ◆。在训练过程中，生成器会不断尝试通过生成更好的假图像来骗过判别器，而判别器在这过程中也会逐步提升判别能力。这种博弈的平衡点是，当生成器生成的假图像和训练数据图像的分布完全一致时，判别器拥有50%的真假判断置信度。

[1] https://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf

GAN

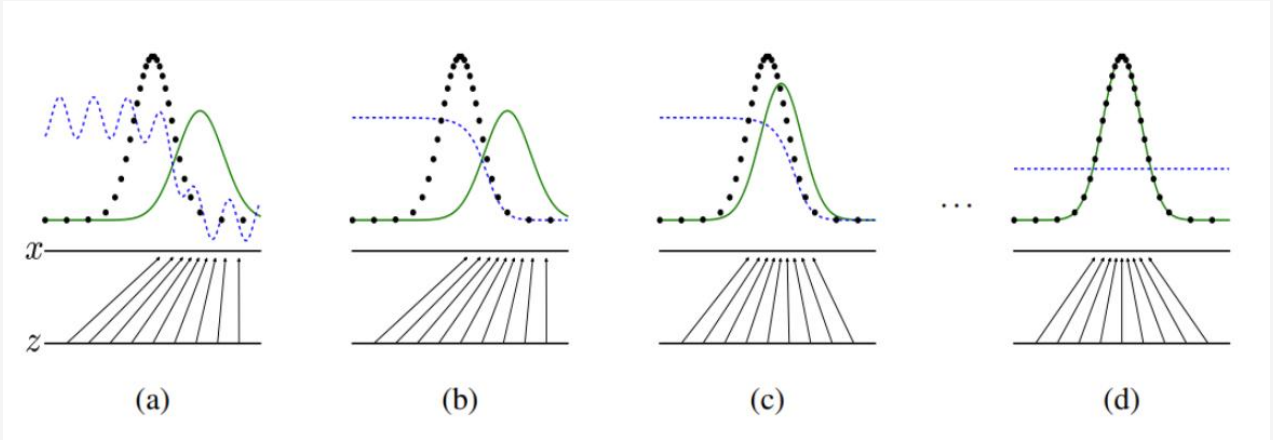
- 判别器D(x)：

x 代表图像数据，用D(x)表示判别器网络给出图像判定为真实图像的概率。在判别过程中，当x来自训练数据时， D(x) 数值应该趋近于1；而当 ❓来自生成器时，D(x) 数值应该趋近于 0。它的作用是判断输入图像 x 是来自真实数据分布 $p_{data}(x)$ 还是来自生成器生成的数据分布 $p_G(x)$，因此 D(x)也可以被当作二分类器。



- 生成器G(z)

G(z) 是一个神经网络，称为生成器。它的作用是将输入的噪声向量 z 映射到数据空间，生成一个数据样本 x。生成器的目标是生成与真实数据分布 $p_{data}(x)$相似的数据分布 $p_G(x;\theta)$。θ是生成网络的参数，训练生成器的目的是找到一组参数最优的 θ。

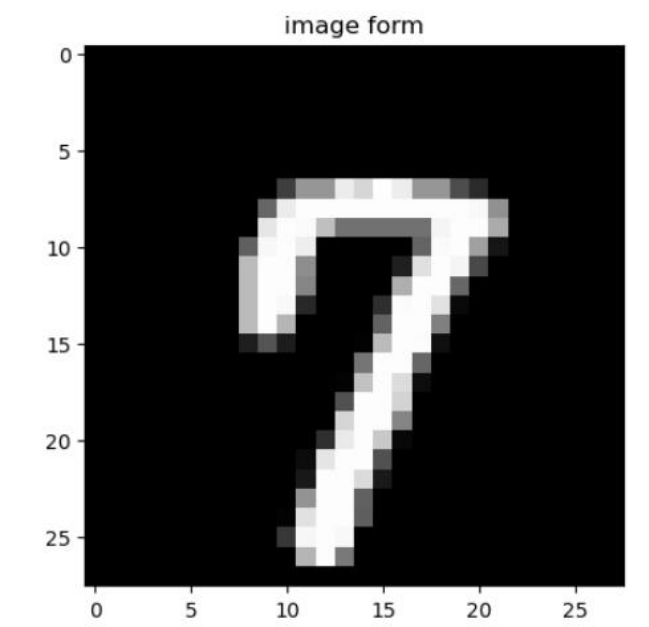


蓝色虚线表示判别器，黑色虚线表示真实数据分布，绿色实线表示生成器生成的虚假数据分布，z是噪声，x 表示生成的虚假图像G（z）。该图片来源于原论文。

用GAN生成手写字

上节课已经下载了MNIST数据集，共有70000张手写数字图片，包含60000张训练样本和10000张测试样本，数字图片为二进制文件，图片大小为28*28，单通道。图片已经预先进行了尺寸归一化和中心化处理。数据解压后的结构如下：

```
./MNIST_Data/
├─ train
│  ├─ train-images-idx3-ubyte
│  └─ train-labels-idx1-ubyte
└─ test
   ├─ t10k-images-idx3-ubyte
   └─ t10k-labels-idx1-ubyte
```



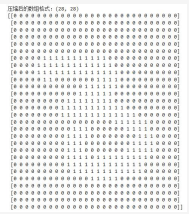
matplotlib.pyplot.imshow 可以将一个二维数组（灰度图像）或者三维数组（例如彩色图像）绘制成可视化为图像。二维数组形状维（H,W），三维数组形状维（H,W,3）（RGB格式），（H,W,4）(RGBA格式)。

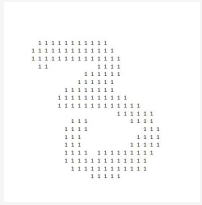
数据加载（对照脚本看）

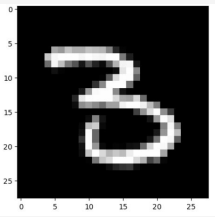
使用MindSpore自己的MnistDatase接口，可以读取和解析MNIST数据集的源文件构建数据集。

加载的数据集有两列: [image, label]。 image 列的数据类型为uint8的数组，是图像的二维矩阵（HWC结构）。 label 列的数据类型为uint32。*试着跑一下脚本，用plot_image()绘制图像。*

这样的数据集合在之前的机器学习训练和神经网络训练时够的，但是现在多出来一个变量 z，这个在论文中被称为latent code的“潜空间码”（我自己叫它宝藏密码，或许你们会给出更好的翻译），GAN的一个核心元素，它可以连接生成器和判别器。所以，我们给每一个图片image，分配一个z，这样每个z也会对应一个image。我们的数据就从（image,label）变成（image,z）。*参考 data_load 函数。*







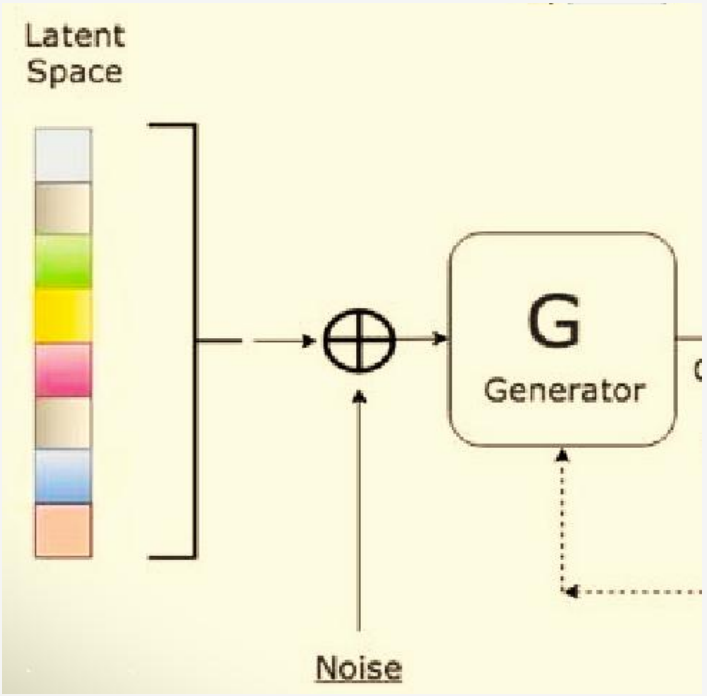
```
def data_load(dataset):
    dataset1 = ds.GeneratorDataset(dataset,
                                   ["image", "label"],
                                   shuffle=True,
                                   python_multiprocessing=False)
    # 数据增强
    mnist_ds = dataset1.map(
        operations=lambda x: (x.astype("float32"),
                              np.random.normal(size=latent_size).astype("float32")),
        output_columns=["image", "latent_code"])
    mnist_ds = mnist_ds.project(["image", "latent_code"])

    # 批量操作
    mnist_ds = mnist_ds.batch(batch_size, True)

    return mnist_ds

mnist_ds = data_load(train_dataset)
```

什么是 latent code

Latent code 是数据在低维空间中的表示（或者你说他是一个特征也行），通常用于生成模型（如 GAN、VAE）或特征提取。

这个向量的维度越高，模型的表达能力越强，能够捕捉更复杂的特征，模型能够捕捉更多的细节和特征，生成样本的质量和多样性较高。但是计算资源消耗更大，甚至在原数据较少的情况下过拟合。

在GAN里，他是生成器的输入，通常是一个随机向量，遵循某种分布（如高斯分布）；同时也是判别器的依据。



为什么分batch

我们的 train_dataset 包含了60000张图片，一般不会将所有图片一次加载到内从中，而是对数据划分批次，形成一个一个小batch。这样可以减少内存占用，提升计算效率。在做梯度下降计算的时候，允许在每个幂次上更新损失，也可以实现小步快跑的效果。另外，一般还有一种假设，所有数据应该是符合同样的数据分布，这样，小批量的数据结论应该是和整体数据的结论是相似的，防止使用全量数据时，出现过拟合。

因此，我们划分128张图片放在一个批次中，一共468个批次。细心的你会发现：128*468=59904，比60000少了96张。这些图片去哪了？mnist_ds = mnist_ds.batch(batch_size, True)，第二参数：True，把这96张图片舍弃了。

模型构建（参考脚本）

论文是最好的参考。这里仿照原论文中代码的 GAN 结构。

因为MNIST 为单通道小尺寸图片，可识别参数少，便于训练，我们在判别器和生成器中采用全连接网络架构和 `ReLU` 激活函数即可，且省略了原论文中用于减少参数的 `Dropout` 策略和可学习激活函数 `Maxout`。

生成器

它的目标是将隐码映射到图像空间。就是创建与真实图像大小相同的灰度图像。这里通过五层 Dense 全连接层来完成的，每层都与 BatchNorm1d 批归一化层和 ReLU 激活层配对，输出数据会经过 Tanh 函数，使其返回 [-1,1] 的数据范围内。

判别器

它是一个二分类网络模型，输出判定该图像为真实图的概率。通过一系列的 Dense 层和 LeakyReLU 层对其进行处理，最后通过 Sigmoid 激活函数，使其返回 [0, 1] 的数据范围内，得到最终概率。

生成器

```python
from mindspore import nn
import mindspore.ops as ops

img_size = 28  # 训练图像长（宽）

class Generator(nn.Cell):
    def __init__(self, latent_size, auto_prefix=True):
        super(Generator, self).__init__(auto_prefix=auto_prefix)
        self.model = nn.SequentialCell()
        # [N, 100] -> [N, 128]
        # 输入一个100维的0~1之间的高斯分布，然后通过第一层线性变换将其映射到256维
        self.model.append(nn.Dense(latent_size, 128))
        self.model.append(nn.ReLU())
        # [N, 128] -> [N, 256]
        self.model.append(nn.Dense(128, 256))
        self.model.append(nn.BatchNorm1d(256))
        self.model.append(nn.ReLU())
        # [N, 256] -> [N, 512]
        self.model.append(nn.Dense(256, 512))
        self.model.append(nn.BatchNorm1d(512))
        self.model.append(nn.ReLU())
        # [N, 512] -> [N, 1024]
        self.model.append(nn.Dense(512, 1024))
        self.model.append(nn.BatchNorm1d(1024))
        self.model.append(nn.ReLU())
        # [N, 1024] -> [N, 784]
        # 经过线性变换将其变成784维
        self.model.append(nn.Dense(1024, img_size * img_size))
        # 经过Tanh激活函数是希望生成的假的图片数据分布能够在-1~1之间
        self.model.append(nn.Tanh())

    def construct(self, x):
        img = self.model(x)
        return ops.reshape(img, (-1, 1, 28, 28))

net_g = Generator(latent_size)
net_g.update_parameters_name('generator')
```

生成器是一个神经网络，由多个全连接层（Dense）、批归一化层（BatchNorm1d）和激活函数（ReLU、Tanh）组成。它的输入是一个随机隐码z，输出是一张图像。

self.model：使用 SequentialCell 定义一个顺序模型容器，表示所有层按顺序堆叠。

第一层：全连接层 + ReLU 激活层
将输入的 100 维z向量映射到 128 维。使用 ReLU 激活函数引入非线性。

第二层：全连接层 + 批归一化 + ReLU 激活，将128维向量映射到256维。BatchNorm1d 对输出进行批归一化。再使用 ReLU 激活函数。

停5分钟，搜一搜全连接层和批归一化层的作用。

非线性激活函数（如 ReLU、Sigmoid、Tanh）的作用是引入非线性，使神经网络能够学习复杂的非线性映射，逼近任意复杂的函数。

第五层也很有趣，我们一起看一下。

模型中的层

Dense 层（全连接层）

对输入数据进行线性变换，可以压缩或者扩展数据。

$$y = Wx + b$$

提示：搜索Dense是如何实现的，体会batch_size的作用。

例如：nn.Dense(100, 128)，表示输入是100维的向量，输出是128维的向量。

BatchNorm1d 层（一维批归一化层）

对输入数据进行归一化，使其均值为 0，方差为 1。

$$\hat{x} = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

其中ϵ 是一个很小的数，用于防止除零错误。

据说：归一化后的数据分布更稳定，可以加快模型的收敛速度；数据分布更平滑，有助于缓解梯度消失或梯度爆炸问题；具有一定的正则化效果，可以减少过拟合。

Tanh 层

双曲正切函数激活层。

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

它的输出范围是[−1,1]，形状类似于 Sigmoid 函数，但 Tanh 的输出以 0 为中心。

输出图像的值范围通过 tanh 激活函数限制在 [-1, 1] 之间。

生成器的前向传播函数

```
def construct(self, x):
    img = self.model(x)
    return ops.reshape(img, (-1, 1, 28, 28))
```

根据 x 生成一个 img，img是一个批次的784维向量。将这个 img 重新变形为： (batch_size, 1, 28, 28) 的形状，即一批 28x28 的单通道图像。

想一想：x 是什么形状?

形状为 (batch_size, latent_size) 的隐空间向量（很多地方叫噪声向量、随机向量）。

判别器

```
# 判别器
class Discriminator(nn.Cell):
    def __init__(self, auto_prefix=True):
        super().__init__(auto_prefix=auto_prefix)
        self.model = nn.SequentialCell()
        # 输入特征数为784，输出为512
        # [N, 784] -> [N, 512]
        self.model.append(nn.Dense(img_size * img_size, 512))
        self.model.append(nn.LeakyReLU())
        # 默认斜率为0.2的非线性映射激活函数
        # [N, 512] -> [N, 256]
        self.model.append(nn.Dense(512, 256))  # 进行一个线性映射
        self.model.append(nn.LeakyReLU())
        # [N, 256] -> [N, 1]
        self.model.append(nn.Dense(256, 1))
        # 二分类激活函数，将实数映射到[0,1]
        self.model.append(nn.Sigmoid())

    def construct(self, x):
        x_flat = ops.reshape(x, (-1, img_size * img_size))
        return self.model(x_flat)

net_d = Discriminator()
net_d.update_parameters_name('discriminator')
```

判别器 是一个二分类网络模型，输出判定该图像为真实图的概率。主要通过一系列的 Dense 层和 LeakyReLU 层对其进行处理，最后通过 Sigmoid 激活函数，使其返回 [0, 1] 的数据范围内，得到最终概率。

有了认识生成器网络的经验，这个结构应该不难。

在 ReLU 中，当输入为负时，输出为 0，梯度也为 0。这可能导致某些神经元永远无法被激活，称为“神经元死亡”问题。

LeakyReLU 在负输入时引入了一个小的斜率 α，使得负输入也有一个非零的梯度，从而缓解了“神经元死亡”问题。

损失函数和优化器

```
# 损失函数
adversarial_loss = nn.BCELoss(reduction='mean')

# 优化器
optimizer_d = nn.Adam(net_d.trainable_params(), learning_rate=lr, beta1=0.5, beta2=0.999)
optimizer_g = nn.Adam(net_g.trainable_params(), learning_rate=lr, beta1=0.5, beta2=0.999)
optimizer_g.update_parameters_name('optim_g')
optimizer_d.update_parameters_name('optim_d')
```

损失函数，用于衡量预测值与真实值差异的程度。MindSpore的损失函数全部是Cell的子类实现，内置的损失函数有：L1Loss、MSELoss、SmoothL1Loss、SoftmaxCrossEntropyWithLogits、SampledSoftmaxLoss、BCELoss 和 CosineEmbeddingLoss。给定一组预测值和真实值，调用损失函数，就可以得到这组预测值和真实值之间的差异。

$$L(y, \hat{y}) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

```
loss = BCELoss() # 定义交叉熵损失函数
input_data = Tensor(**) # 模预测值
target_data = Tensor(**) # 模拟真实值
output = loss(input_data, target_data) # 模拟差异
```

优化器在模型训练过程中，用于计算和更新网络参数，可以有效减少训练时间，提高最终模型性能。最基本的优化器是梯度下降（SGD），在此基础上，很多其他的优化器进行了改进，以实现目标函数能更快速更有效地收敛到全局最优点。优化器能够保持当前参数状态并基于计算得到的梯度进行参数更新。要指定需要优化的网络权重（必须是Parameter实例），然后设置参数选项，比如学习率，权重衰减等。

模型训练

训练分为两部分：

1、训练判别器。训练判别器的目的是最大程度地提高判别图像真伪的概率。按照原论文的方法，通过提高其随机梯度来更新判别器，最大化 log𝐷(x)+log(1−𝐷(𝐺(z)) 的值。

2、训练生成器。如论文所述，最小化 log(1−𝐷(𝐺(z))) 来训练生成器，以产生更好的虚假图像。

```
import os
import time
import matplotlib.pyplot as plt
import mindspore as ms
from mindspore import Tensor, save_checkpoint

total_epoch = 200  # 训练周期数
batch_size = 128  # 用于训练的训练集批量大小

# 加载预训练模型的参数
pred_trained = False
pred_trained_g = './result/checkpoints/Generator99.ckpt'
pred_trained_d = './result/checkpoints/Discriminator99.ckpt'

checkpoints_path = "./result/checkpoints"  # 结果保存路径
image_path = "./result/images"  # 测试结果保存路径
```

训练开始前，一些常规操作：
1、定义total_epoch：总训练轮次；
2、可以加载别人与训练过的参数；
3、可以让每轮训练的模型都输出一些生成图片，让我们可以看到生成的进化过程。

模型训练：预定义函数

def generator_forward(test_noises):

- 使用生成器 net_g 将随机噪声 test_noises 映射为假数据 fake_data。
- 使用判别器 net_d 对假数据 fake_data 进行判别，得到输出 fake_out。
- 使用二分类交叉熵损失函数 adversarial_loss 计算生成器的损失。
- 目标是让判别器对假数据的输出 fake_out 接近 1（即欺骗判别器）。

def discriminator_forward(real_data, test_noises):

- 使用生成器 net_g 将随机噪声 test_noises 映射为假数据 fake_data。
- 使用判别器 net_d 对假数据 fake_data 进行判别，得到输出 fake_out。
- 使用判别器 net_d 对真实数据 real_data 进行判别，得到输出 real_out。
- 使用二分类交叉熵损失函数 adversarial_loss 计算真实数据的损失 real_loss 和假数据的损失 fake_loss，这两相加是总的损失loss_d。
- 优化的目标是让判别器对真实数据的输出 real_out 接近 1，对假数据的输出 fake_out 接近 0。

模型训练：预定义函数

grad_g = ms.value_and_grad(generator_forward, None, net_g.trainable_params())

grad_d = ms.value_and_grad(discriminator_forward, None, net_d.trainable_params())

- 计算生成器和判别器的损失值及梯度，用于更新模型参数。

def train_step(real_data, latent_code):

- 使用 grad_d 计算判别器的损失 loss_d 和梯度 grads_d。
- 使用优化器 optimizer_d 更新判别器的参数。
- 使用 grad_g 计算生成器的损失 loss_g 和梯度 grads_g。
- 使用优化器 optimizer_g 更新生成器的参数。
- 返回判别器和生成器的损失值，用于监控训练过程。

通过交替更新生成器和判别器的参数，实现 GAN 的训练。

模型训练：开始练

```python
for epoch in range(total_epoch):
    start = time.time()
    for (iter, data) in enumerate(mnist_ds):
        start1 = time.time()
        image, latent_code = data
        image = (image - 127.5) / 127.5  # [0, 255] -> [-1, 1]
        image = image.reshape(image.shape[0], 1, image.shape[1], image.shape[2])
        d_loss, g_loss = train_step(image, latent_code)
        end1 = time.time()
        if iter % 10 == 0:
            print(f"Epoch:[{int(epoch):>3d}/{int(total_epoch):>3d}], "
                  f"step:[{int(iter):>4d}/{int(iter_size):>4d}], "
                  f"loss_d:{d_loss.asnumpy():>4f} , "
                  f"loss_g:{g_loss.asnumpy():>4f} , "
                  f"time:{(end1 - start1):>3f}s, "
                  f"lr:{lr:>6f}")

    end = time.time()
    print("time of epoch {} is {:.2f}s".format(epoch + 1, end - start))

    losses_d.append(d_loss.asnumpy())
    losses_g.append(g_loss.asnumpy())

    # 每个epoch结束后，使用生成器生成一组图片
    gen_imgs = net_g(test_noise)
    save_imgs(gen_imgs.asnumpy(), epoch)

    # 根据epoch保存模型权重文件
    if epoch % 1 == 0:
        save_checkpoint(net_g, checkpoints_path + "/Generator%d.ckpt" % (epoch))
        save_checkpoint(net_d, checkpoints_path + "/Discriminator%d.ckpt" % (epc
```

根据预定义轮次，开始循环。

取出当前的batch，包含了image，latent_code，对image做归一化，并且重新变形。

对这一batch进行训练，获取d_loss,g_loss，屏幕输出：

```
Epoch:[  0/200], step:[   0/ 468], loss_d:1.319662 , loss_g:0.719872 , time:0.29
Epoch:[  0/200], step:[  10/ 468], loss_d:1.127259 , loss_g:0.669279 , time:0.05
Epoch:[  0/200], step:[  20/ 468], loss_d:1.318720 , loss_g:0.638649 , time:0.05
Epoch:[  0/200], step:[  30/ 468], loss_d:1.331956 , loss_g:0.760032 , time:0.18
Epoch:[  0/200], step:[  40/ 468], loss_d:1.273975 , loss_g:0.852013 , time:0.13
Epoch:[  0/200], step:[  50/ 468], loss_d:1.356580 , loss_g:0.908828 , time:0.11
```

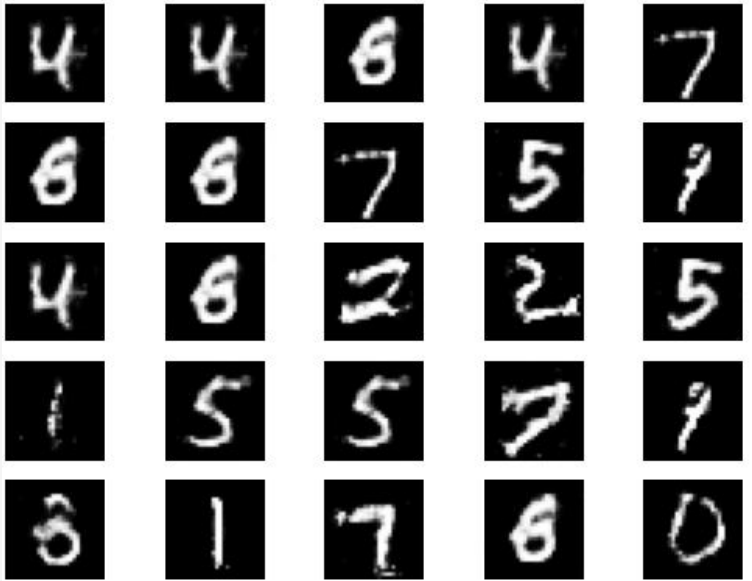
提示：一轮（epoch）大约需要1分半，200轮还需要很多时间滴~

结果

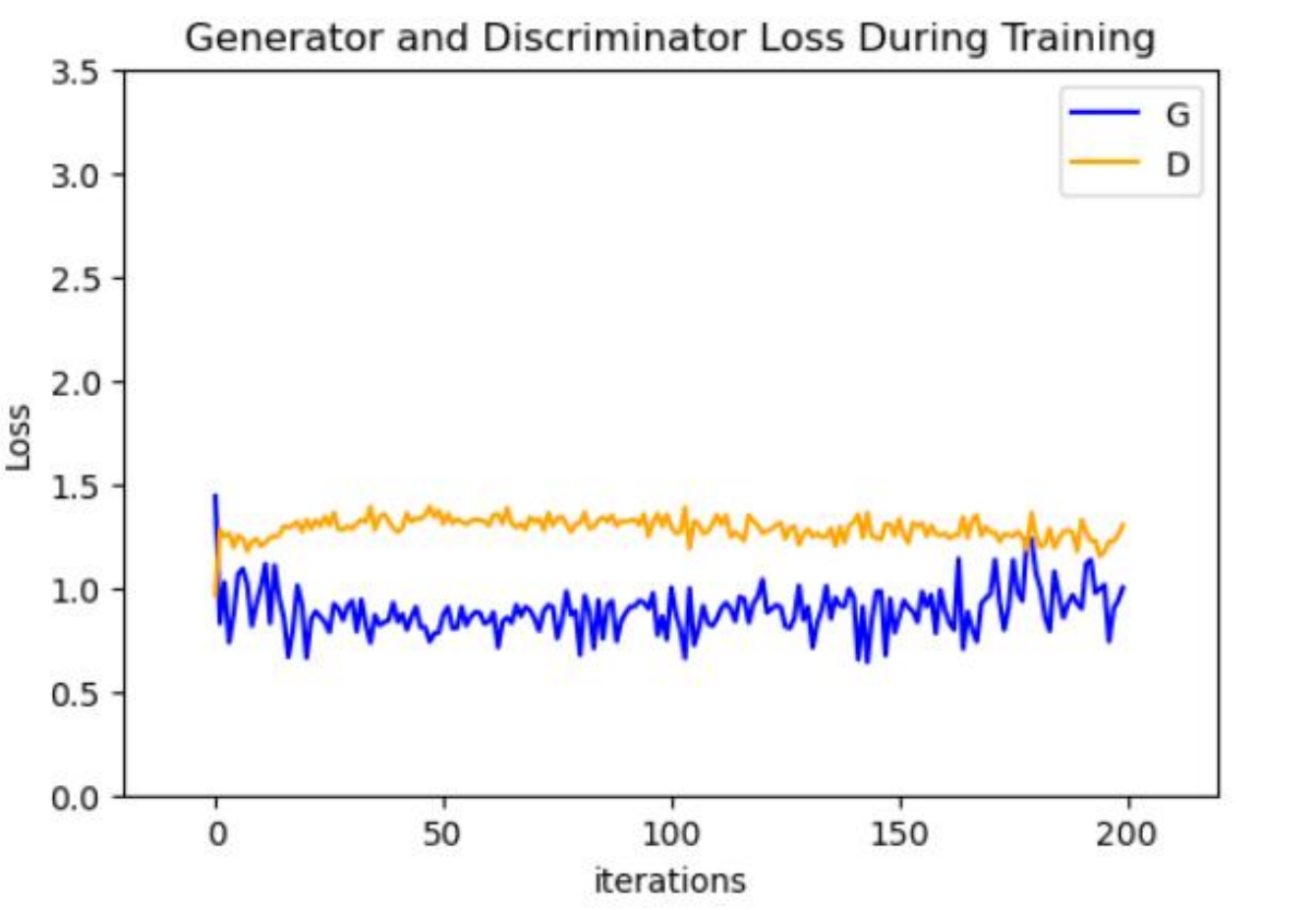
还记得我们前面定义了一组25个随机向量么?

[4]:
```
import random
import numpy as np
from mindspore import Tensor
from mindspore.common import dtype

# 利用随机种子创建一批隐码
np.random.seed(2323)
test_noise = Tensor(np.random.normal(size=(25, 100)), dtype.float32)
random.shuffle(test_noise)
```

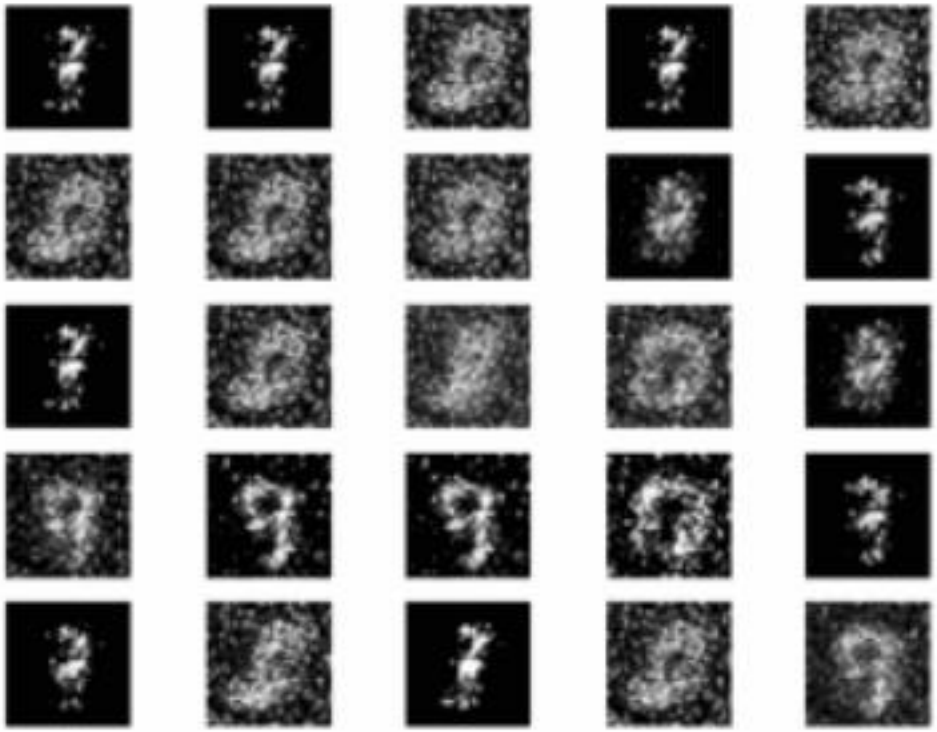
这25个向量生成的图形就是下面的这些，这当然是第200轮训练后的结果:



再看看训练过程中的损失函数



动态看一下训练后的效果



接下来

模型训练好了，就应该将模型用起来，怎么用?

- 生成器：给他一个向量数据让他生成28*28像素的图片；（搜索一下mindspore的评估模式和训练模式）
- 判别器：给他一组28*28的像素图片，让他输出这个图片是生成的图片。
- 修改超参数：
  - epoch 总的训练轮次：（可以配合已经训练出来的参数继续训练，一轮大概1分多种，不要设置太多）
  - 学习率：如果学习率设置很大，会怎么样？（能不能动态学习率）
  - 批次大小：想一想为什么常用32，64，128，256。
  - 隐码向量维度：现在是100，更大或者更小会怎么样?
  - 优化器参数：现在使用的Adam优化器，beta1一阶矩估计的置属衰减率，beta2二阶矩估计的指数衰减率。
  - 损失函数：BCELoss的参数。
  - 网络结构、激活函数（这些可能需要从头进行训练）

MS例子：DCGAN

GAN还能不能进化

DCGAN（深度卷积对抗生成网络，Deep Convolutional Generative Adversarial Networks）是GAN的直接扩展。不同之处在于，DCGAN会分别在判别器和生成器中使用卷积和转置卷积层。

它最早由Radford等人在论文 Unsupervised Representation Learning With Deep Convolutional Generative Adversarial Networks[1] 中进行描述。判别器由分层的卷积层、BatchNorm层和LeakyReLU激活层组成。输入是3x64x64的图像，输出是该图像为真图像的概率。生成器则是由转置卷积层、BatchNorm层和ReLU激活层组成。输入是标准正态分布中提取出的隐向量z，输出是3x64x64的RGB图像。

[1]https://arxiv.org/pdf/1511.06434.pdf

之前GAN我们改了参数，但……

西安交通大学 XI'AN JIAOTONG UNIVERSITY

我么们换个数据集，MS官方的下载渠道：

https://download.mindspore.cn/dataset/Faces/faces.zip

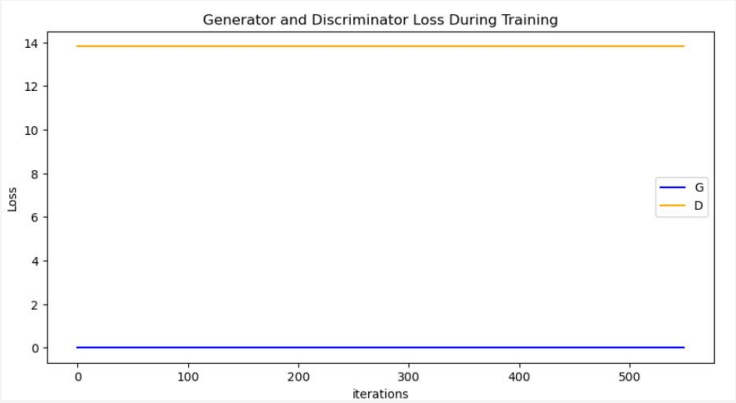
动漫头像数据集来训练一个生成式对抗网络，使用该网络生成动漫头像图片。

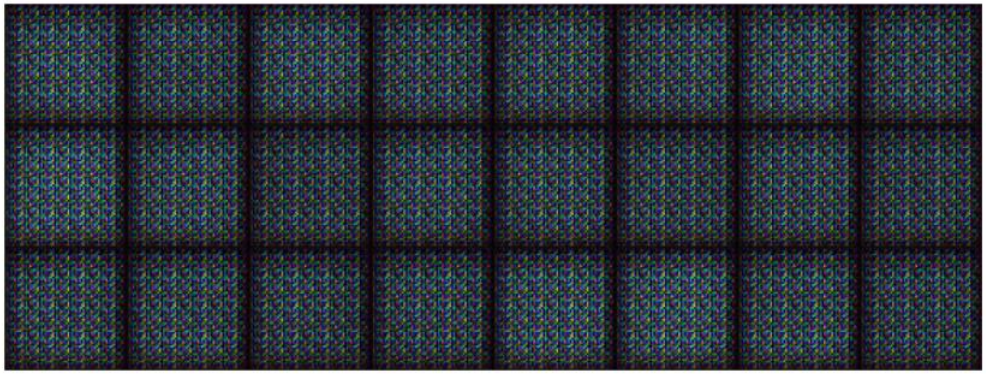
跑一跑，改一改

这下面的结果咋看着不对呢？没有人保证说有的参数一把就能出结果。

```
Epoch:[  1/  2], step:[   1/ 275], loss_d:13.815511 , loss_g:-0.000000 , time:0.001012s
Epoch:[  1/  2], step:[  51/ 275], loss_d:13.815511 , loss_g:-0.000000 , time:0.001483s
Epoch:[  1/  2], step:[ 101/ 275], loss_d:13.815511 , loss_g:-0.000000 , time:0.000000s
Epoch:[  1/  2], step:[ 151/ 275], loss_d:13.815511 , loss_g:-0.000000 , time:0.000998s
Epoch:[  1/  2], step:[ 201/ 275], loss_d:13.815511 , loss_g:-0.000000 , time:0.000000s
Epoch:[  1/  2], step:[ 251/ 275], loss_d:13.815511 , loss_g:-0.000000 , time:0.000995s
Epoch:[  1/  2], step:[ 275/ 275], loss_d:13.815510 , loss_g:-0.000000 , time:2.441543s
time of epoch 1 is 1110.89s
Epoch:[  2/  2], step:[   1/ 275], loss_d:13.815511 , loss_g:-0.000000 , time:0.001089s
Epoch:[  2/  2], step:[  51/ 275], loss_d:13.815511 , loss_g:-0.000000 , time:0.001996s
Epoch:[  2/  2], step:[ 101/ 275], loss_d:13.815511 , loss_g:-0.000000 , time:0.001000s
Epoch:[  2/  2], step:[ 151/ 275], loss_d:13.815511 , loss_g:-0.000000 , time:0.000997s
Epoch:[  2/  2], step:[ 201/ 275], loss_d:13.815511 , loss_g:-0.000000 , time:0.002005s
Epoch:[  2/  2], step:[ 251/ 275], loss_d:13.815511 , loss_g:-0.000000 , time:0.001011s
Epoch:[  2/  2], step:[ 275/ 275], loss_d:13.815510 , loss_g:-0.000000 , time:0.007095s
time of epoch 2 is 938.62s
```

想想怎么试一试。





西安交通大学
XI'AN JIAOTONG UNIVERSITY

谢谢大家

什么是交叉熵

熵是用来描述一个系统的混乱程度，交叉熵就能够表示预测数据与真实数据的相近程度。预测结果可以认为是一个在不同结果类型上的一个概率分布，交叉熵越小，表示预测结果的概率分布越接近真实样本的分布。

$$L\ (y,\hat{y})\ =\ -[y\ log(\hat{y}) + (1-y)log(1-\hat{y})]$$

y 表示真实标签：0或1

$\hat{y}$ 表示模型预测的概率值

对数函数 log可以将概率值映射到一个更敏感的尺度上，使得小概率事件对损失的贡献更大。

如果真实值是1，预测概率为0.1，则损失为：$-1 * log\ (0.1)\ \approx 2.30$

如果真实值是1，预测概率为0.9，则损失为：$-1 * log\ (0.9)\ \approx 0.11$

当模型预测的概率与真实标签相差较大时，交叉熵损失会显著增加。

这两篇博客中的例子可能更易懂：

- https://www.cnblogs.com/Fish0403/p/17073047.html
- https://zhuanlan.zhihu.com/p/638725320

标题

西安交通大学
XI'AN JIAOTONG UNIVERSITY