7

生成式对抗网络（GANs)

在本章中，我们将研究生成式对抗网络（GANs）[1]。GANs 属于生成模型家族。然而，与自动编码器不同的是，生成模型能够在给定的任意编码中创建新的、有意义的输出。在这一章中，我们将讨论 GANs 的工作原理。我们还将回顾几个使用 tf.keras 的早期 GANs 的实现，而在本章后面，我们将展示实现稳定训练所需的技术。本章的范围包括两个流行的 GAN 实现实例，深度卷积 GAN（DCGAN）[2] 和条件 GAN（CGAN）[3]。综上所述，本章的目标是：- 介绍 GAN 的原理 - 介绍 GAN 的早期工作实现之一，称为 DCGAN - 一个改进的 DCGAN，称为 CGAN，它使用一个条件 - 在 tf.keras 中实现 DCGAN 和 CGAN 让我们先来看看 GAN 的概况。

7.1. GANs 的概述

1. An Overview of GANs

在我们进入 GANs 的更高级的概念之前，让我们先了解一下 GANs 并介绍其背后的基本概念。GANs 是非常强大的；这个简单的声明被以下事实所证明：它们可以通过执行潜伏空间插值来生成非真人的新人脸。GANs 的高级功能可以在这些 YouTube 视频中看到。- Progressive GAN [4]: https://youtu.be/G06dEcZ-QTg - StyleGAN v1 [5]: https://youtu.be/kSLJriaOumA - StyleGAN v2 [6]: https://youtu.be/c-NJtV9Jvp0 这些视频展示了如何利用 GANs 来制作逼真的人脸，展示了它们可以有多么强大。这个主题比我们之前在本书中看到的任何东西都要高级得多。例如，上述视频展示了自动编码器无法轻易完成的事情，我们在第 3 章自动编码器中介绍过。GANs 能够通过训练两个相互竞争（和合作）的网络来学习如何对输入分布进行建模，这两个网络被称为生成器和鉴别器（有时被称为批判器）。生成器的作用是不断找出如何生成假数据或信号（这包括音频和图像），以骗过判别器。同时，鉴别器被训练来区分假的和真的信号。随着训练的进行，鉴别器将不再能够看到合成的数据和真实数据之间的区别。从此，鉴别器可以被丢弃，然后生成器就可以用来创建以前从未观察到的新的现实数据。GANs 的基本概念是简单明了的。然而，我们会发现一个问题，即最具挑战性的问题是如何实现生成器-判别器网络的稳定训练？为了使两个网络能够同时学习，生成器和鉴别器之间必须存在良性竞争。由于损失函数是从鉴别器的输出中计算出来的，它的参数更新很快。当鉴别器的收敛速度较快时，生成器的参数就不再有足够的梯度更新，从而无法收

致。除了难以训练之外，GANs 还可能出现部分或全部模式崩溃的情况，即生成器对不同的潜伏编码产生几乎相似的输出。使用 www.DeepL.com/Translator 翻译（免费版）

7.1.1. GAN 的原理

如图7.1所示，GAN 类似于造假者（产生者）-警察（辨别者）的情景。在学院里，警察被教导如何确定一张美元钞票是真的还是假的。来自银行的真钞和来自造假者的假钞样本被用来训练警察。然而，造假者会不时地试图假装他印制了真正的美元钞票。起初，警察不会上当，会告诉造假者为什么钱是假的。考虑到这一反馈，造假者会再次磨练自己的技能，并试图制作新的假美元钞票。正如预期的那样，警察既能发现这些钱是假的，也能证明为什么这些美元钞票是假的。

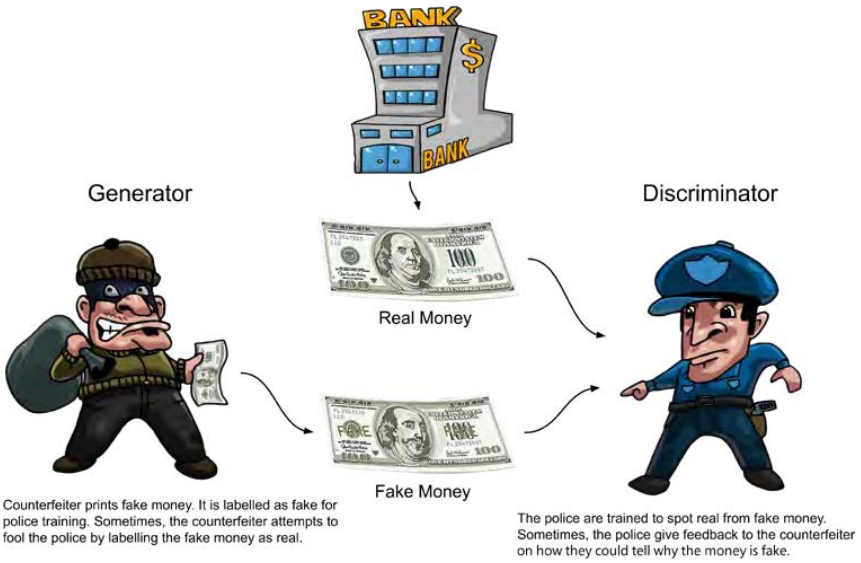

图 7.1: GANs 的生成者和辨别者类似于造假者和警察。造假者的目标是欺骗警察，使其相信美元钞票是真的。

这个过程无止境地继续下去，但它会发展到一个地步，即造假者已经掌握了制造假币的技术，以至于假币与真币无法区分–即使是对最有经验的警察来说。然后，造假者可以无限地印制美元钞票而不被警察抓到，因为它们不再能被识别为假钞。如图7.2所示，GAN 由两个网络组成，一个生成器和一个鉴别器。

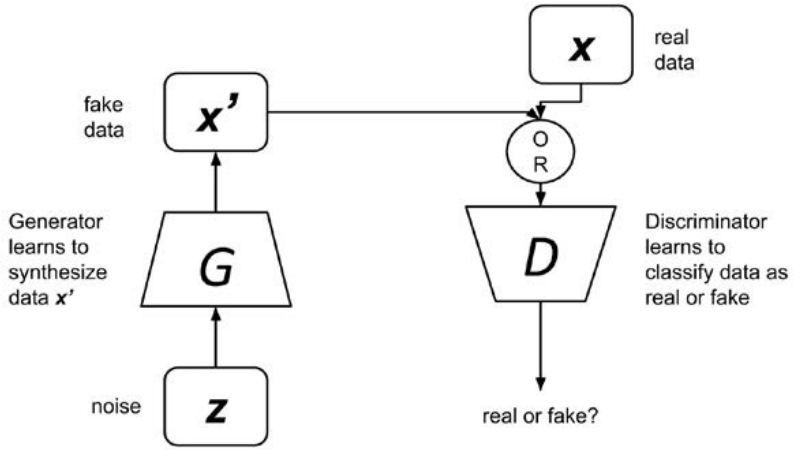

图 7.2: GAN 由两个网络组成，一个是生成器，一个是鉴别器。鉴别器被训练来区分真实和虚假的信号或数据。

生成器的输入是噪声，而输出是合成数据。同时，鉴别器的输入将是真实的或合成的数据。真数据来自于真实的采样数据，而假数据则来自于发生器。所有的有效数据都被标记为 1.0（即 100% 的真实概率），而所有的合成数据被标记为 0.0（即 0% 的真实概率）。由于标注过程是自动化的，GANs 仍然被认为是深度学习中无监督学习方法的一部分。鉴别器的目标是从这个提供的数据集中学习如何区分真实数据和虚假数据。在 GAN 训练的这一部分，只有判别器的参数会被更新。像典型的二进制分类器一样，判别器被训练成在 0.0 到 1.0 的范围内预测给定输入数据与真实数据的接近程度。然而，这仅仅是故事的一半。每隔一段时间，生成器会假装其输出是真实数据，并要求 GAN 将其标记为 1.0。当假数据被提交给鉴别器时，它自然会被分类为假数据，标签接近 0.0。

优化器根据呈现的标签（即 1.0）计算生成器的参数更新。在对新数据进行训练时，它也考虑到了自己的预测。这个新数据时，它也会考虑自己的预测。换句话说，鉴别器对它的预测有一些怀疑，所以，GAN 会考虑到这一点。这一次，GAN 将让梯度从鉴别器的最后一层反向传播到发生器的第一层。然而，在大多数实践中，在训练的这个阶段，判别器的参数被暂时冻结。生成器将使用梯度来更新其参数并提高其合成假数据的能力。总的来说，整个过程类似于两个网络在相互竞争的同时还在进行合作。当 GAN 训练收敛时，最终的结果是一个能够合成看起来真实的数据的发生器。鉴别器认为这个合成的数据是真实的或者标签接近 1.0，这意味着鉴别器就可以被抛弃了。生成器部分将有助于从任意的噪声输入中产生有意义的输出。该过程在下面的图**??**中概述。

$, compute discriminator gradients$

6.w $\leftarrow w - \alpha \times RMSProp(w, g_w)$, update discriminator parameters

7. $w \leftarrow clip(w, -c, c)$, clip discriminator weights

8. end for

9. Sample a batch $\{z^{(i)}\}_{i=1^m\ p(z)}$ from uniform noise distribution

10. $g_\Theta \leftarrow -\nabla_\Theta \frac{1}{m} \sum_{i=1}^{m} D_w(g_\Theta(z^{(i)}))$, compute generator gradients

11. $\Theta \leftarrow \Theta - \alpha \times RMSProp(\Theta, g_\Theta)$,update generator parameters

12. end while

图9.3说明，除了假/真 (fake/true) 数据标签和损失函数，WGAN 模型实际上与 DCGAN 相同

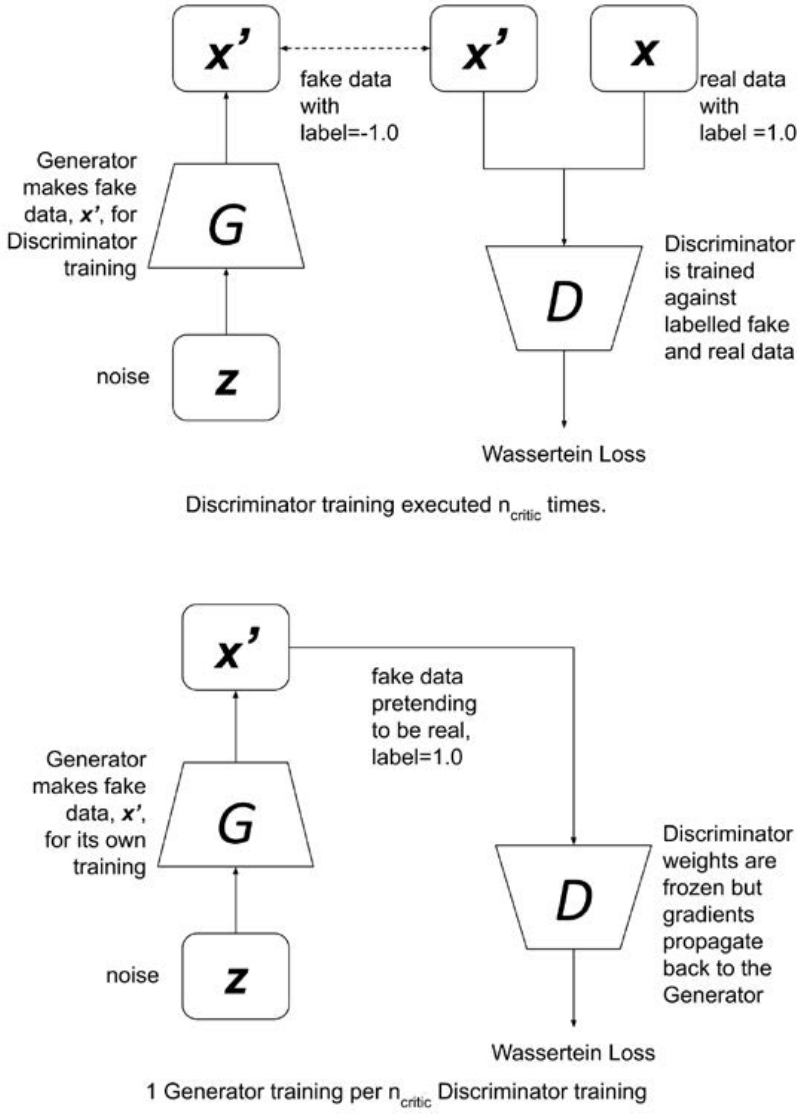

图 8.3: 上半部分：训练 WGAN 判别器需要来自生成器的假数据和来自真实分布的真实数据。下半部分：训练 WGAN 生成器需要来自生成器的假数据，假装是真实的。

与 GANs 类似，WGAN 交替训练判别器和生成器（通过对抗）。然而，在 WGAN 中，鉴别器（也称为批评者）在训练生成器的一次迭代（第 9 至 11 行）之前，先训练 n 次关键的迭代（第 2 至 8 行）。这与判别器和生成器的训练迭代次数相同的 GANs 形成对比。换句话说，在 GANs 中，$n_{critic} = 1$。

训练鉴别器意味着学习鉴别器的参数（权重和偏差）。这需要从真实数据（第 3 行）和虚假数据（第 4 行）中抽出一批，并在将抽样数据送入判别器网络后计算判别器参数的梯度（第 5 行）。使用 RMSProp 优化鉴别器参数（第 6 行）。第 5 行和第 6 行都是对方程**??**的优化。

最后，EM 距离优化中的 Lipschitz 约束是通过剪切判别器参数来实施的（第 7 行）。第 7 行是方程**??**的执行。在判别器训练的 $n_{critic} = 1$ 迭代之后，判别器参数被冻结。生成器的训练开始于

对一批假数据的采样（第 9 行）。采样的数据被标记为真实的（1.0），努力欺骗鉴别器网络。第 10 行计算发生器梯度，第 11 行使用 RMSProp 进行优化。第 10 行和第 11 行进行梯度更新以优化方程**??**。

训练完生成器后，判别器的参数被解冻，另一个 $n_{critic} = 1$ 判别器的训练迭代开始。我们应该注意到，在判别器训练期间没有必要冻结生成器的参数，因为生成器只参与数据的制造。与 GANs 类似，判别器可以作为一个单独的网络进行训练。然而，训练生成器总是需要鉴别器通过对抗性网络参与，因为损失是由生成器网络的输出计算出来的。

与 GAN 不同，在 WGAN 中，真实数据被标记为 1.0，而假数据被标记为-1.0，作为第 5 行计算梯度的一种变通方法。第 5-6 行和第 10-11 行分别进行梯度更新以优化方程**??**和**??**。第 5 行和第 10 行中的每个项都被建模为公式9.33：

$$L = -y_{label}\frac{1}{m}\sum_{i=1}^{m} y_{pred} \tag{8.33}$$

其中，真实数据的 $y_{label} = 1.0$，虚假数据的 $y_{label} = -1.0$。为了简化符号，我们删除了上标 (i)。对于判别器，WGAN 增加 □ 以使使用真实数据训练时的损失函数最小。

当使用假数据进行训练时，WGAN 减少 $y_{pred} = D_w(g(z))$ 以使损失函数最小。对于生成器，WGAN 增加 $y_{pred} = D_w(g(z))$，以便在训练期间将假数据标记为真数据时使损失函数最小。请注意，y_{label} 在损失函数中除了其符号外没有直接的贡献。在 tf.keras 中，方程**??**被实现为。

def wasserstein_loss(y_label, y_pred): return -K.mean(y_label * y_pred)

本节最重要的部分是用于 GANs 稳定训练的新损失函数。该算法正式确定了 WGAN 的完整训练算法，包括损失函数。在下一节中，将介绍该训练算法在 tf.keras 中的实现。

8.1.4. 使用 Keras 实现 WGAN

为了在 tf.keras 中实现 WGAN，我们可以重用 GAN 的 DCGAN 实现，这是我们在上一章中介绍的。DCGAN 构建器和实用函数作为一个模块在 lib 文件夹中的 gan.py 中实现。这些函数包括 - generator(): 一个生成器模型的建立者 - discriminator(): 鉴别器模型构建器 - train(): 一个 DCGAN 训练器 - plot_images(): 一个通用的生成器的输出绘图器 - test_generator(): 一个通用的生成器测试工具如清单 5.1.1 所示，我们可以通过简单地调用：discriminator = gan.discriminator(inputs, activation='linear') 建立一个判别器。WGAN 使用线性输出激活。对于生成器，我们执行 generator = gan.generator(inputs, image_size)tf.keras 中的整体网络模型类似于图**??**中的 DCGAN。下列程序强调了 RMSprop 优化器和 Wasserstein 损失函数的使用。函数。算法 5.1.1 中的超参数在训练过程中被使用。[2]

Listing 8.1: RMSprop 优化器和 Wasserstein 损失函数的使用

```
Listing 5.1.1: wgan-mnist-5.1.2.py
   def build_and_train_models():
       """Load the dataset, build WGAN discriminator,
       generator, and adversarial models.
       Call the WGAN train routine.
       """
       # load MNIST dataset
       (x_train, _), (_, _) = mnist.load_data()
       # reshape data for CNN as (28, 28, 1) and normalize
```

[2]The complete code is available on GitHub: https://github. com/PacktPublishing/Advanced-Deep-Learning-with- Keras

```
        image_size = x_train.shape[1]
        x_train = np.reshape(x_train, [-1, image_size, image_size, 1])
        x_train = x_train.astype('float32') / 255
        model_name = "wgan_mnist"
# network parameters
# the latent or z vector is 100-dim latent_size = 100
# hyper parameters from WGAN paper [2] n_critic = 5
clip_value = 0.01
batch_size = 64
lr = 5e-5
train_steps = 40000
input_shape = (image_size, image_size, 1)
# build discriminator model
inputs = Input(shape=input_shape, name='discriminator_input') # WGAN uses linear activation in
    paper [2]
discriminator = gan.discriminator(inputs, activation='linear') optimizer = RMSprop(lr=lr)
# WGAN discriminator uses wasserstein loss discriminator.compile(loss=wasserstein_loss,
                              optimizer=optimizer,
                              metrics=['accuracy'])
discriminator.summary()
# build generator model
        input_shape = (latent_size, )
        inputs = Input(shape=input_shape, name='z_input')
        generator = gan.generator(inputs, image_size)
        generator.summary()
        # build adversarial model = generator + discriminator
        # freeze the weights of discriminator during adversarial training
        discriminator.trainable = False
        adversarial = Model(inputs,
                            discriminator(generator(inputs)),
                            name=model_name)
        adversarial.compile(loss=wasserstein_loss,
                            optimizer=optimizer,
                            metrics=['accuracy'])
        adversarial.summary()
        # train discriminator and adversarial networks
        models = (generator, discriminator, adversarial)
        params = (batch_size,
                  latent_size,
                  n_critic,
                  clip_value,
                  train_steps,
                  model_name)
        train(models, x_train, params)
```

程序 5.1.2 是紧跟算法 5.1.1 的训练函数。然而，在训练判别器时有一个小小的调整。我们将先用一批真实数据进行训练，然后再用一批虚假数据进行训练，而不是用一批真实和虚假数据来训练权重。这一调整将防止梯度消失，因为真实数据和虚假数据的标签符号是相反的，而且由于剪切，权重的幅度很小。

Listing 8.2: Evaluate how the model does on the test set

```
Listing 5.1.2: wgan-mnist-5.1.2.py Training algorithm for WGAN:
def train(models, x_train, params):
"""Train the Discriminator and Adversarial Networks Alternately train Discriminator and Adversarial
     networks by batch.
```

```
Discriminator is trained first with properly labelled real and fake images for n_critic times.
    Discriminator weights are clipped as a requirement
of Lipschitz constraint.
Generator is trained next (via Adversarial) with
fake images pretending to be real.
Generate sample images per save_interval
Arguments:
    models (list): Generator, Discriminator,
        Adversarial models
    x_train (tensor): Train images
    params (list) : Networks parameters
"""
# the GAN models
generator, discriminator, adversarial = models
# network parameters
(batch_size, latent_size, n_critic,
        clip_value, train_steps, model_name) = params
# the generator image is saved every 500 steps
save_interval = 500
# noise vector to see how the
# generator output evolves during training
noise_input = np.random.uniform(-1.0,
1.0,
                                size=[16, latent_size])
# number of elements in train dataset
train_size = x_train.shape[0]
# labels for real data
real_labels = np.ones((batch_size, 1))
for i in range(train_steps):
    # train discriminator n_critic times
    loss = 0
    acc = 0
    for _ in range(n_critic):
        # train the discriminator for 1 batch
        # 1 batch of real (label=1.0) and
        # fake images (label=-1.0)
        # randomly pick real images from dataset
        rand_indexes = np.random.randint(0,
train_size,
                                         size=batch_size)
        real_images = x_train[rand_indexes]
        # generate fake images from noise using generator
        # generate noise using uniform distribution
        noise = np.random.uniform(-1.0,
1.0,
                                  size=[batch_size, latent_size])
        fake_images = generator.predict(noise)
        # train the discriminator network
        # real data label=1, fake data label=-1
        constraint
# instead of 1 combined batch of real and fake images,
# train with 1 batch of real data first, then 1 batch
# of fake images.
# this tweak prevents the gradient
# from vanishing due to opposite
# signs of real and fake data labels (i.e. +1 and -1) and
# small magnitude of weights due to clipping.
```

```
real_loss, real_acc = \
    discriminator.train_on_batch(real_images,
                                 real_labels)
fake_loss, fake_acc = \
    discriminator.train_on_batch(fake_images,
                                 -real_labels)
# accumulate average loss and accuracy
loss += 0.5 * (real_loss + fake_loss)
acc += 0.5 * (real_acc + fake_acc)
# clip discriminator weights to satisfy Lipschitz
for layer in discriminator.layers:
    weights = layer.get_weights()
    weights = [np.clip(weight,
                       -clip_value,
                       clip_value) for weight in weights]
    layer.set_weights(weights)
# average loss and accuracy per n_critic training iterations
loss /= n_critic
acc /= n_critic
log = "%d: [discriminator loss: %f, acc: %f]" % (i, loss, acc)
# train the adversarial network for 1 batch
# 1 batch of fake images with label=1.0
# since the discriminator weights are frozen in
# adversarial network only the generator is trained
# generate noise using uniform distribution
noise = np.random.uniform(-1.0,
1.0,
                          size=[batch_size, latent_size])
# train the adversarial network
# note that unlike in discriminator training,
# we do not save the fake images in a variable
# the fake images go to the discriminator
# input of the adversarial for classification
# fake images are labelled as real
# log the loss and accuracy
loss, acc = adversarial.train_on_batch(noise, real_labels)
            log = "%s [adversarial loss: %f, acc: %f]" % (log, loss, acc)
            print(log)
            if (i + 1) % save_interval == 0:
                # plot generator images on a periodic basis
                gan.plot_images(generator,
                                noise_input=noise_input,
                                show=False,
                                step=(i + 1),
                                model_name=model_name)
        # save the model after training the generator
        # the trained generator can be reloaded
        # for future MNIST digit generation
        generator.save(model_name + ".h5")
```

图9.4显示了 WGAN 在 MNIST 数据集上输出的演变。

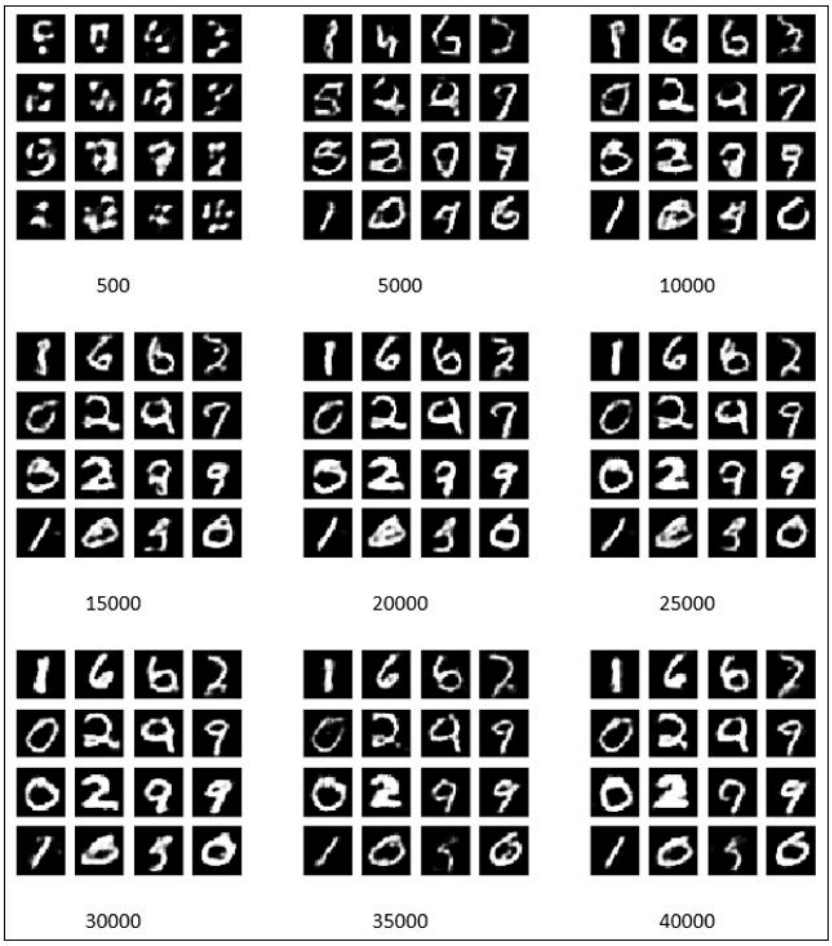

图 8.4: WGAN 的样本输出与训练步骤的对比。在训练和测试期间，WGAN 的任何输出都没有出现模式崩溃。

即使在网络配置变化下，WGAN 也是稳定的。例如，已知 DCGAN 在判别器网络的 ReLU 之前插入批量归一化时是不稳定的。同样的配置在 WGAN 中是稳定的。下面的图9.5向我们展示了 DCGAN 和 WGAN 在判别器网络中批量归一化的输出。

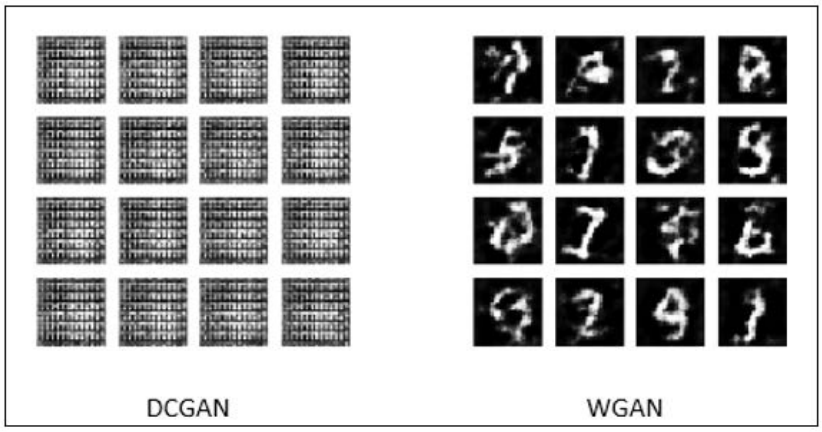

图 8.5: 在判别器网络的 ReLU 激活之前插入批量标准化时，DCGAN（左）和 WGAN（右）的输出比较

与前一章的 GAN 训练类似，训练后的模型在 4 万个训练步骤后被保存在一个文件中。使用训练好的生成器模型，通过运行以下命令生成新的合成 MNIST 数字图像。

python3 wgan-mnist-5.1.2.py –generator=wgan_mnist.h5

正如我们所讨论的，原始的 GAN 是很难训练的。当 GAN 优化其损失函数时，问题就出现了；它实际上是在优化 JS 发散，D_{JS}。当两个分布函数之间几乎没有重叠时，就很难优化 D_{JS}。WGAN 提出通过使用 EMD 或 Wasserstein 1 损失函数来解决这个问题，即使在两个分布之间很少或没有重叠的情况下，它也有一个平滑的可微分函数。然而，WGAN 并不关注生成的图像质量。除了稳定性问题，原始 GAN 的生成图像在感知质量方面仍有需要改进的地方。LSGAN 的理论是，这两个问题可以同时解决。我们将在下一节看一下 LSGAN。

8.2. 最小二乘法 GAN (LSGAN)

LSGAN 提出了最小二乘法的损失。图9.6展示了为什么在 GANs 中使用 sigmoid 交叉熵损失会导致生成的数据质量很差。

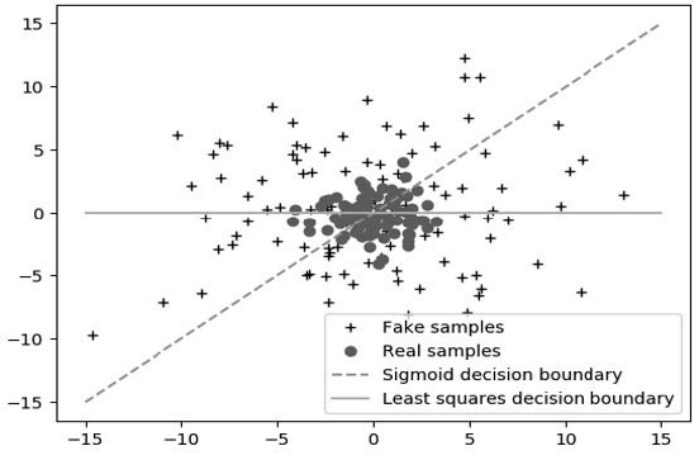

图 **8.6:** Both real and fake sample distributions divided by their respective decision boundaries: sigmoid and least squares

理想情况下，虚假样本的分布应该尽可能地接近真实样本的分布。然而，对于 GANs 来说，一旦假样本已经在决策边界的正确一侧，梯度就会消失。这使得生成器没有足够的动力来提高生成的假数据的质量。远离决策边界的虚假样本将不再试图向真实样本的分布靠近。使用最小二乘损失函数，只要假样本分布远离真实样本分布，梯度就不会消失。即使假样本已经在决策边界的正确一侧，生成器也会努力改善其对真实密度分布的估计。

最小化方程9.6或判别器损失函数意味着真实数据分类和真实标签 1.0 之间的 MSE 应该接近于零。此外，假数据分类和真实标签 0.0 之间的 MSE 应该接近于零。

与其他 GAN 类似，LSGAN 判别器被训练成从假数据样本中分类出真实数据。最小化方程9.7意味着在标签 1.0 的帮助下，欺骗鉴别器，使其认为生成的假数据样本是真的。

使用前一章中的 DCGAN 代码作为基础来实现 LSGAN 只需要做一些改变。如程序 5.2.1 所示，鉴别器的 sigmoid 激活被删除。鉴别器是通过调用来建立的。

Listing 8.3: 建立鉴别器

```
discriminator = gan.discriminator(inputs, activation=None)
```

该发生器与原来的 DCGAN 类似。

Listing 8.4: 建立发生器

```
generator = gan.generator(inputs, image_size)
```

鉴别器和对抗性损失函数都由 mse 代替。所有的网络参数都与 DCGAN 相同。tf.keras 中 LSGAN 的网络模型与图**??**相似，只是有线性或没有输出激活。训练过程与 DCGAN 中看到的类似，由效用函数提供：

Listing 8.5: Evaluate how the model does on the test set

```
gan.train(models, x_train, params)
```

Listing 8.6: Evaluate how the model does on the test set

```
Listing 5.2.1: lsgan-mnist-5.2.1.py
```

```
    def build_and_train_models():
        """Load the dataset, build LSGAN discriminator,
        generator, and adversarial models.
        Call the LSGAN train routine.
        """
        # load MNIST dataset
        (x_train, _), (_, _) = mnist.load_data()
        # reshape data for CNN as (28, 28, 1) and normalize
image_size = x_train.shape[1]
x_train = np.reshape(x_train,
                     [-1, image_size, image_size, 1])
x_train = x_train.astype('float32') / 255
model_name = "lsgan_mnist"
# network parameters
# the latent or z vector is 100-dim
latent_size = 100
input_shape = (image_size, image_size, 1)
batch_size = 64
lr = 2e-4
decay = 6e-8
train_steps = 40000
# build discriminator model
inputs = Input(shape=input_shape, name='discriminator_input') discriminator = gan.discriminator(
    inputs, activation=None)
# [1] uses Adam, but discriminator easily
# converges with RMSprop
optimizer = RMSprop(lr=lr, decay=decay)
# LSGAN uses MSE loss [2]
discriminator.compile(loss='mse',
                      optimizer=optimizer,
                      metrics=['accuracy'])
discriminator.summary()
# build generator model
input_shape = (latent_size, )
inputs = Input(shape=input_shape, name='z_input')
generator = gan.generator(inputs, image_size)
generator.summary()
# build adversarial model = generator + discriminator
optimizer = RMSprop(lr=lr*0.5, decay=decay*0.5)
# freeze the weights of discriminator
# during adversarial training
discriminator.trainable = False
adversarial = Model(inputs,
                    discriminator(generator(inputs)),
                    name=model_name)
# LSGAN uses MSE loss [2]
adversarial.compile(loss='mse',
                    optimizer=optimizer,
                    metrics=['accuracy'])
adversarial.summary()
# train discriminator and adversarial networks
models = (generator, discriminator, adversarial)
       params = (batch_size, latent_size, train_steps, model_name)
       gan.train(models, x_train, params)
```

图9.7显示了使用 MNIST 数据集进行 40,000 步训练后生成的 LSGAN 样本。

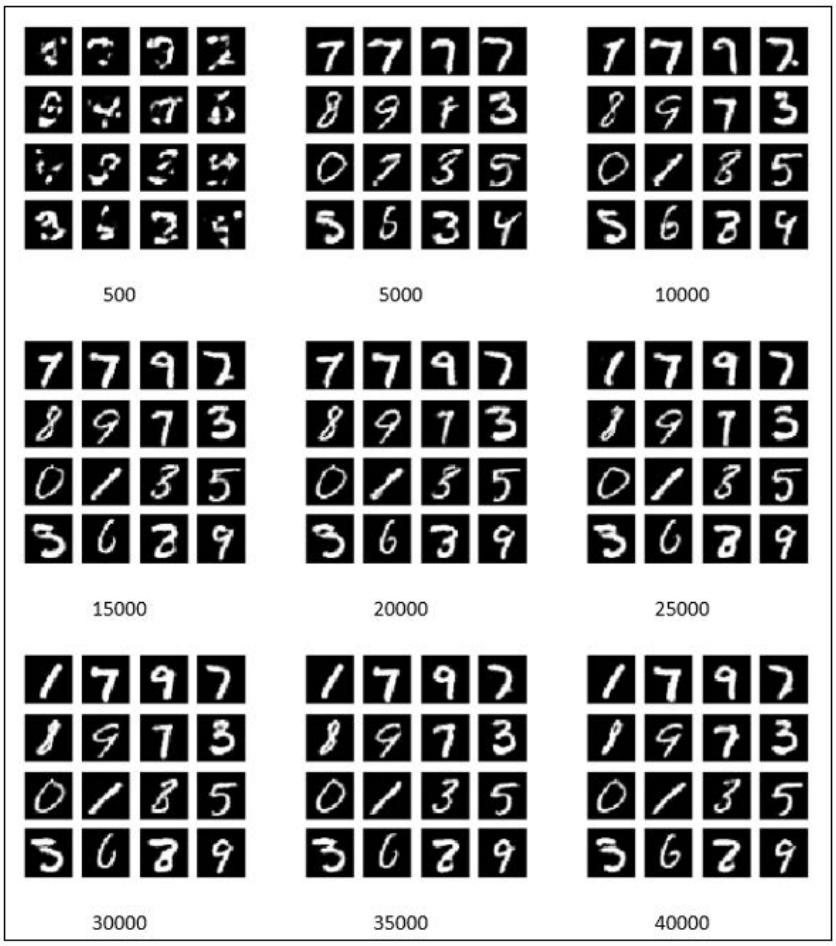

图 8.7: LSGAN 的样本输出与训练步骤的对比

与前一章看到的 DCGAN 中的图**??**相比，输出的图像具有更好的感知质量。使用训练好的生成器模型，通过运行以下命令生成新的合成的 MNIST 数字图像。

Listing 8.7: Evaluate how the model does on the test set

```
python3 lsgan-mnist-5.2.1.py --generator=lsgan_mnist.h5
```

在本节中，我们讨论了损失函数的另一个改进。通过使用 MSE 或 L2，我们解决了训练 GANs 的稳定性和感知质量这两个问题。在下一节中，我们提出了另一项改进，这次是与 CGAN 有关，这在上一章中已经讨论过。

8.3. 辅助分类器 GAN（ACGAN）

ACGAN 在原理上类似于我们在上一章讨论的条件性 GAN（CGAN）。我们将比较 CGAN 和 ACGAN。对于 CGAN 和 ACGAN，生成器的输入是噪声和它的标签。输出是一个属于输入类别标签的假图像。对于 CGAN，判别器的输入是图像（假的或真的）和它的标签。输出是该图像是真的概率。对于 ACGAN，鉴别器的输入是一幅图像，而输出是该图像是真实的概率，其类别是一个标

签。

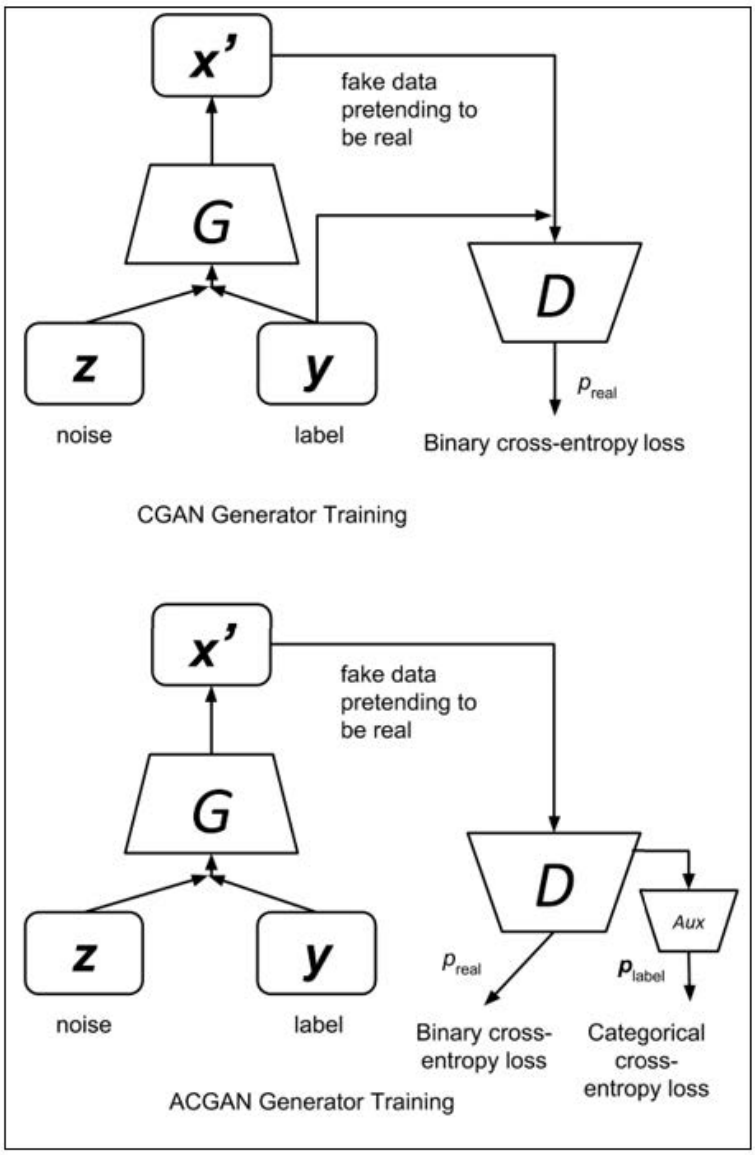

图 8.8: CGAN 与 ACGAN 发生器的训练。主要的区别是判别器的输入和输出

基本上，在 CGAN 中，我们用侧面信息（标签）来喂养网络。在 ACGAN 中，我们试图用一个辅助类解码器网络来重构侧面信息。ACGAN 理论认为，强迫网络做额外的任务是已知的，以提高原始任务的性能。在这种情况下，额外的任务是图像分类。原始任务是生成假图像。

除了额外的分类器损失函数，ACGAN 的损失函数与 CGAN 相同。除了识别真假图像的原始任务 $(-\mathbb{E}_{x\sim p_{data}} \log D(x|y) - \mathbb{E}_z \log(1 - D(g(z|y))))$，方程式9.8 鉴别器的有一个额外的任务，即正确地对真假图像进行分类 $(-\mathbb{E}_{x\sim p_{data}} \log p(c|x) - \mathbb{E}_z \log(c|g(z|y))$。方程9.9的生成器意味着，除了试图用假图像来欺骗鉴别器 $(-\mathbb{E}_z \log D(g(x|y)))$ 以外。它要求判别器正确分类那些假图像

$(-\mathbb{E}_z \log p(c|g(z|y)))$。

从 CGAN 代码开始，只有判别器和训练函数被修改以实现 ACGAN。鉴别器和生成器构建函数也由 gan.py 提供。为了看到在鉴别器上所做的修改，程序 5.3.1 显示了构建器函数，其中执行图像分类的辅助解码器网络和双输出被突出显示。

Listing 8.8: Listing 5.3.1: gan.py

```
Listing 5.3.1: gan.py
   def discriminator(inputs,
                     activation='sigmoid',
                     num_labels=None,
                     num_codes=None):
       """Build a Discriminator Model
       Stack of LeakyReLU-Conv2D to discriminate real from fake
       The network does not converge with BN so it is not used here
       unlike in [1]
       Arguments:
inputs (Layer): Input layer of the discriminator (the image) activation (string): Name of output
     activation layer num_labels (int): Dimension of one-hot labels for ACGAN &
   InfoGAN
           num_codes (int): num_codes-dim Q network as output
                       if StackedGAN or 2 Q networks if InfoGAN
       Returns:
           Model: Discriminator Model
       """
       kernel_size = 5
       layer_filters = [32, 64, 128, 256]
       x = inputs
       for filters in layer_filters:
           # first 3 convolution layers use strides = 2
           # last one uses strides = 1
           if filters == layer_filters[-1]:
               strides = 1
           else:
               strides = 2
           x = LeakyReLU(alpha=0.2)(x)
           x = Conv2D(filters=filters,
                      kernel_size=kernel_size,
                      strides=strides,
                      padding='same')(x)
       x = Flatten()(x)
       # default output is probability that the image is real
       outputs = Dense(1)(x)
       if activation is not None:
           print(activation)
           outputs = Activation(activation)(outputs)
       if num_labels:
           # ACGAN and InfoGAN have 2nd output
           # 2nd output is 10-dim one-hot vector of label
           layer = Dense(layer_filters[-2])(x)
           labels = Dense(num_labels)(layer)
           labels = Activation('softmax', name='label')(labels)
           if num_codes is None:
               outputs = [outputs, labels]
               else:
               # InfoGAN have 3rd and 4th outputs
               # 3rd output is 1-dim continous Q of 1st c given x
```

```
                code1 = Dense(1)(layer)
                code1 = Activation('sigmoid', name='code1')(code1)
                # 4th output is 1-dim continuous Q of 2nd c given x
                code2 = Dense(1)(layer)
                code2 = Activation('sigmoid', name='code2')(code2)
                outputs = [outputs, labels, code1, code2]
        elif num_codes is not None:
            # StackedGAN Q0 output
            # z0_recon is reconstruction of z0 normal distribution
            z0_recon =  Dense(num_codes)(x)
            z0_recon = Activation('tanh', name='z0')(z0_recon)
            outputs = [outputs, z0_recon]
        return Model(inputs, outputs, name='discriminator')
```

然后通过调用建立判别器：

Listing 8.9: Evaluate how the model does on the test set

```
discriminator = gan.discriminator(inputs, num_labels=num_labels)
```

该生成器与 WGAN 和 LSGAN 中的生成器相同。回顾一下，生成器的构建器在下面的程序 5.3.2 中显示。我们应该注意，程序 5.3.1 和 5.3.2 都是 WGAN 和 LSGAN 在前几节中使用的相同的生成器函数。突出显示的是适用于 LSGAN 的部分。

Listing 8.10: Listing 5.3.2: gan.py

```
Listing 5.3.2: gan.py
    def generator(inputs,
                  image_size,
                  activation='sigmoid',
                  labels=None,
                  codes=None):
        """Build a Generator Model
        Stack of BN-ReLU-Conv2DTranpose to generate fake images.
        Output activation is sigmoid instead of tanh in [1].
        Sigmoid converges easily.
        Arguments:
            inputs (Layer): Input layer of the generator (the z-vector)
            image_size (int): Target size of one side
                (assuming square image)
                activation (string): Name of output activation layer
    labels (tensor): Input labels
    codes (list): 2-dim disentangled codes for InfoGAN
Returns:
    Model: Generator Model
"""
image_resize = image_size // 4
# network parameters
kernel_size = 5
layer_filters = [128, 64, 32, 1]
if labels is not None:
    if codes is None:
        # ACGAN labels
        # concatenate z noise vector and one-hot labels
        inputs = [inputs, labels]
    else:
        # infoGAN codes
```

```
        # concatenate z noise vector,
        # one-hot labels and codes 1 & 2
        inputs = [inputs, labels] + codes
    x = concatenate(inputs, axis=1)
elif codes is not None:
    # generator 0 of StackedGAN
    inputs = [inputs, codes]
    x = concatenate(inputs, axis=1)
else:
    # default input is just 100-dim noise (z-code)
    x = inputs
x = Dense(image_resize * image_resize * layer_filters[0])(x)
x = Reshape((image_resize, image_resize, layer_filters[0]))(x)
for filters in layer_filters:
    # first two convolution layers use strides = 2
    # the last two use strides = 1
    if filters > layer_filters[-2]:
        strides = 2
    else:
        strides = 1
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
    x = Conv2DTranspose(filters=filters,
                        kernel_size=kernel_size,
                               strides=strides,
                               padding='same')(x)
       if activation is not None:
           x = Activation(activation)(x)
       # generator output is the synthesized image x
       return Model(inputs, x, name='generator')
```

在 ACGAN 中，发生器被实例化为：

Listing 8.11: 在 ACGAN 中，发生器被实例化程序

```
generator = gan.generator(inputs, image_size, labels=labels)
```

图、9.9显示了 tf.keras 中 ACGAN 的网络模型。

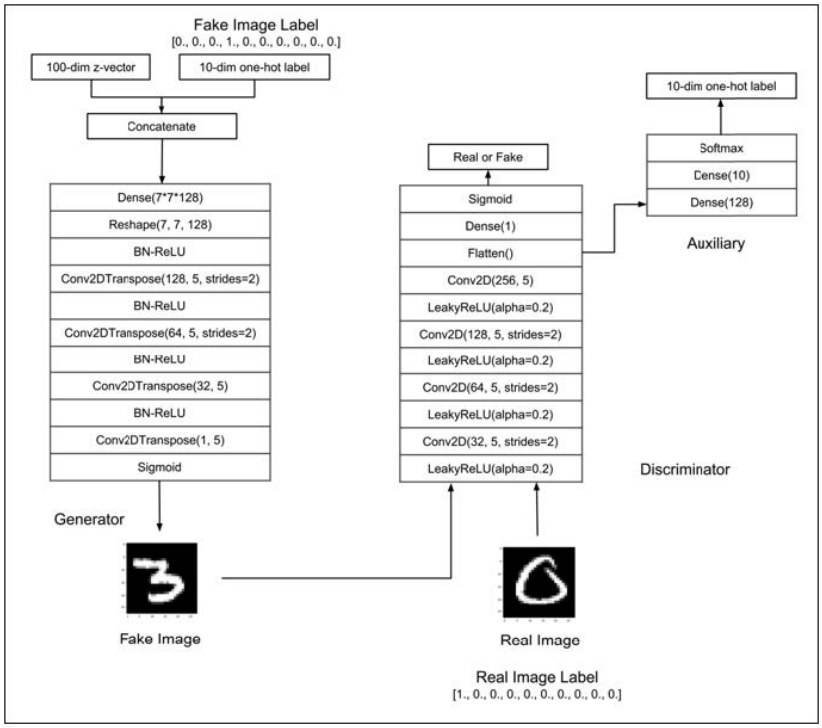

图 **8.9:** ACGAN 的 tf.keras 模型

如程序 5.3.3 所示，鉴别器和对抗性模型被修改以适应鉴别器网络的变化。我们现在有两个损失函数。第一个是原始的二进制交叉熵，用于训练判别器估计输入图像为真实的概率。

第二个是预测类标签的图像分类器。输出是一个 10 维的单热向量。

Listing 5.3.3: acgan-mnist-5.3.1.py

Listing 8.12: Listing 5.3.3: acgan-mnist-5.3.1.py

```
def build_and_train_models():
        """Load the dataset, build ACGAN discriminator,
        generator, and adversarial models.
        Call the ACGAN train routine.
        """
        # load MNIST dataset
        (x_train, y_train), (_, _) = mnist.load_data()
        # reshape data for CNN as (28, 28, 1) and normalize
        image_size = x_train.shape[1]
        x_train = np.reshape(x_train,
                             [-1, image_size, image_size, 1])
        x_train = x_train.astype('float32') / 255
        # train labels
        num_labels = len(np.unique(y_train))
        y_train = to_categorical(y_train)
        model_name = "acgan_mnist"
        # network parameters
        latent_size = 100
        batch_size = 64
        train_steps = 40000
```

```
        lr = 2e-4
        decay = 6e-8
        input_shape = (image_size, image_size, 1)
        label_shape = (num_labels, )
        # build discriminator Model
        inputs = Input(shape=input_shape,
                       name='discriminator_input')
        # call discriminator builder
        # with 2 outputs, pred source and labels
        discriminator = gan.discriminator(inputs,
num_labels=num_labels)
# [1] uses Adam, but discriminator
# easily converges with RMSprop
        optimizer = RMSprop(lr=lr, decay=decay)
        # 2 loss fuctions: 1) probability image is real
        # 2) class label of the image
        loss = ['binary_crossentropy', 'categorical_crossentropy']
        discriminator.compile(loss=loss,
                              optimizer=optimizer,
                              metrics=['accuracy'])
        discriminator.summary()
# build generator model
input_shape = (latent_size, )
inputs = Input(shape=input_shape, name='z_input') labels = Input(shape=label_shape, name='labels')
    # call generator builder with input labels generator = gan.generator(inputs,
                                  image_size,
                                  labels=labels)
        generator.summary()
        # build adversarial model = generator + discriminator
        optimizer = RMSprop(lr=lr*0.5, decay=decay*0.5)
        # freeze the weights of discriminator
        # during adversarial training
        discriminator.trainable = False
        adversarial = Model([inputs, labels],
                            discriminator(generator([inputs, labels])),
                            name=model_name)
        # same 2 loss fuctions: 1) probability image is real
        # 2) class label of the image
        adversarial.compile(loss=loss,
                            optimizer=optimizer,
                            metrics=['accuracy'])
        adversarial.summary()
        # train discriminator and adversarial networks
        models = (generator, discriminator, adversarial)
        data = (x_train, y_train)
        params = (batch_size, latent_size, \
                  train_steps, num_labels, model_name)
        train(models, data, params)
```

在程序 5.3.4 中，我们强调了在训练程序中实现的变化。与 CGAN 代码相比，主要的区别是在判别器和对抗性训练中必须提供输出标签。

Listing 8.13: Listing 5.3.4: acgan-mnist-5.3.1.py

```
%Listing 5.3.4: acgan-mnist-5.3.1.py
   def train(models, data, params):
        """Train the discriminator and adversarial Networks
        Alternately train discriminator and adversarial
```

```
        networks by batch.
        Discriminator is trained first with real and fake
        images and corresponding one-hot labels.
        Adversarial is trained next with fake images pretending
        to be real and corresponding one-hot labels.
        Generate sample images per save_interval.
        # Arguments
            models (list): Generator, Discriminator,
                Adversarial models
            data (list): x_train, y_train data
            params (list): Network parameters
        """
# the GAN models
generator, discriminator, adversarial = models
# images and their one-hot labels
x_train, y_train = data
# network parameters
batch_size, latent_size, train_steps, num_labels, model_name \
                = params
        # the generator image is saved every 500 steps
        save_interval = 500
        # noise vector to see how the generator
        # output evolves during training
        noise_input = np.random.uniform(-1.0,
1.0,
                                        size=[16, latent_size])
# class labels are 0, 1, 2, 3, 4, 5,
# 6, 7, 8, 9, 0, 1, 2, 3, 4, 5
# the generator must produce these MNIST digits
noise_label = np.eye(num_labels)[np.arange(0, 16) % num_labels] # number of elements in train
    dataset
train_size = x_train.shape[0]
print(model_name,
              "Labels for generated images: ",
              np.argmax(noise_label, axis=1))
        for i in range(train_steps):
            # train the discriminator for 1 batch
            # 1 batch of real (label=1.0) and fake images (label=0.0)
            # randomly pick real images and
# corresponding labels from dataset
rand_indexes = np.random.randint(0,
train_size,
                                 size=batch_size)
real_images = x_train[rand_indexes]
real_labels = y_train[rand_indexes]
# generate fake images from noise using generator
# generate noise using uniform distribution
noise = np.random.uniform(-1.0,
1.0,
                          size=[batch_size, latent_size])
# randomly pick one-hot labels
fake_labels = np.eye(num_labels)[np.random.choice(num_labels,
                                                  batch_size)]
# generate fake images
fake_images = generator.predict([noise, fake_labels]) # real + fake images = 1 batch of train data
x = np.concatenate((real_images, fake_images))
# real + fake labels = 1 batch of train data labels labels = np.concatenate((real_labels,
```

```
    fake_labels))
# label real and fake images
# real images label is 1.0
y = np.ones([2 * batch_size, 1])
# fake images label is 0.0
y[batch_size:, :] = 0
# train discriminator network, log the loss and accuracy # ['loss', 'activation_1_loss',
# 'label_loss', 'activation_1_acc', 'label_acc']
metrics = discriminator.train_on_batch(x, [y, labels]) fmt = "%d: [disc loss: %f, srcloss: %f,"
fmt += "lblloss: %f, srcacc: %f, lblacc: %f]"
log = fmt % (i, metrics[0], metrics[1], \
        metrics[2], metrics[3], metrics[4])
# train the adversarial network for 1 batch
# 1 batch of fake images with label=1.0 and
# corresponding one-hot label or class
# since the discriminator weights are frozen
# in adversarial network only the generator is trained
# generate noise using uniform distribution
noise = np.random.uniform(-1.0,
                          1.0,

                                    size=[batch_size, latent_size])
           # randomly pick one-hot labels
           fake_labels = np.eye(num_labels)[np.random.choice(num_labels,
                                                             batch_size)]
# label fake images as real
y = np.ones([batch_size, 1])
# train the adversarial network
# note that unlike in discriminator training,
# we do not save the fake images in a variable
# the fake images go to the discriminator input
# of the adversarial for classification
# log the loss and accuracy
metrics = adversarial.train_on_batch([noise, fake_labels],
                                                  [y, fake_labels])
           fmt = "%s [advr loss: %f, srcloss: %f,"
           fmt += "lblloss: %f, srcacc: %f, lblacc: %f]"
           log = fmt % (log, metrics[0], metrics[1],\
                   metrics[2], metrics[3], metrics[4])
           print(log)
           if (i + 1) % save_interval == 0:
               # plot generator images on a periodic basis
               gan.plot_images(generator,
                           noise_input=noise_input,
                           noise_label=noise_label,
                           show=False,
                           step=(i + 1),
                           model_name=model_name)
       # save the model after training the generator
       # the trained generator can be reloaded
       # for future MNIST digit generation
       generator.save(model_name + ".h5")
```

事实证明，有了额外的任务，与我们之前讨论过的所有 GAN 相比，ACGAN 的性能提升是非常明显的。ACGAN 的训练是稳定的，如图9.10所示，ACGAN 对以下标签的输出样本。

Listing 8.14: ACGAN 的输出样本

```
[0 1 2 3
4 5 6 7
8 9 0 1
2 3 4 5]
```

与 CGAN 不同的是，在训练过程中，样本输出的外观并没有很大的变化。MNIST 数字图像的感知质量也更好。

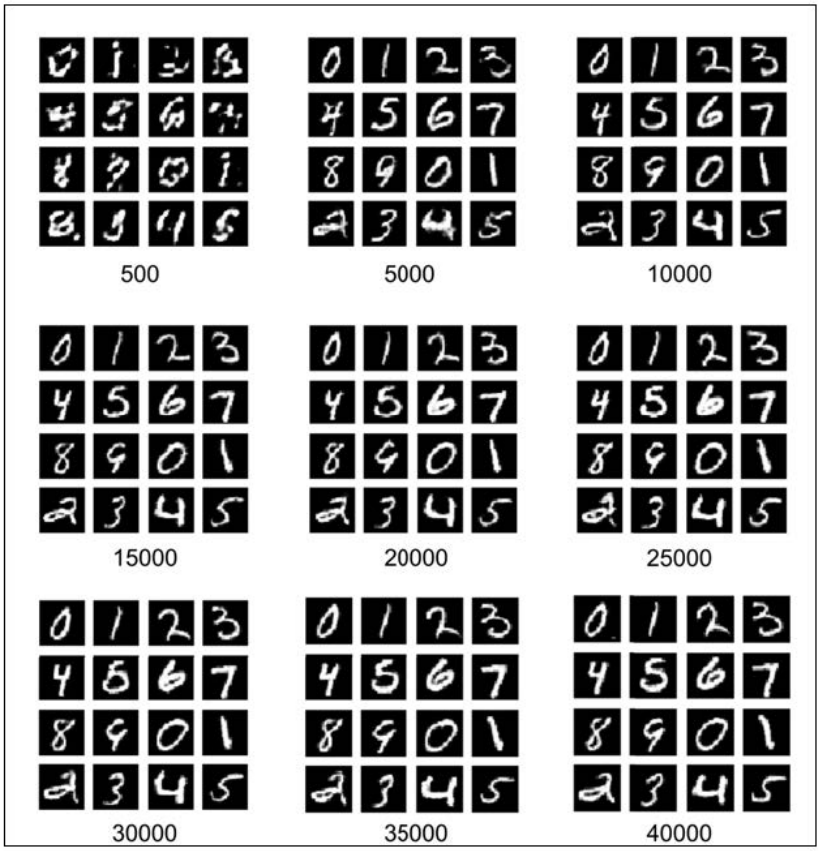

图 8.10: ACGAN 生成的样本输出是标签 [0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5] 的训练步骤的函数。

使用经过训练的生成器模型，通过运行生成新的合成 MNIST 数字图像：python3 acgan-mnist-5.3.1.py –generator=acgan_mnist.h5 另外，也可以要求生成一个特定的数字（例如 3）来生成：python3 acgan-mnist-5.3.1.py –generator=acgan_mnist.h5 –digit=3

图9.11显示了 CGAN 和 ACGAN 产生的 MNIST 数字的并列比较。数字 2-6 在 ACGAN 中的质量比在 CGAN 中要好。

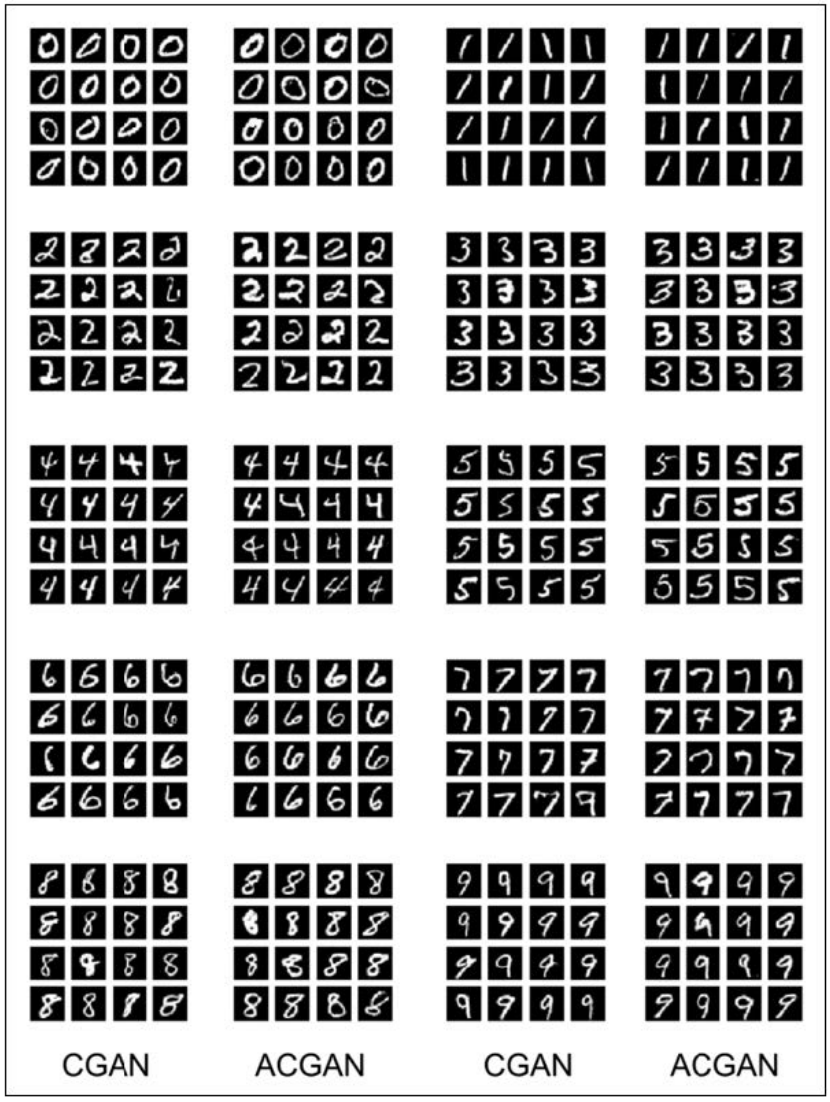

图 8.11: 对 CGAN 和 ACGAN 在数字 0 到 9 的条件下的输出进行并列比较

与 WGAN 和 LSGAN 类似，ACGAN 通过微调其损失函数，对现有的 GAN–CGAN 进行了改进。在接下来的章节中，我们将发现新的损失函数，使 GANs 能够执行新的有用的任务。

I

8.4. 总结

在这一章中，我们介绍了对上一章中首次介绍的原始 GAN 算法的各种改进。WGAN 提出了一种算法，通过使用 EMD 或 Wasserstein 1 损失来提高训练的稳定性。LSGAN 认为，GAN 的原始交叉熵函数容易出现梯度消失，这与最小二乘损失不同。LSGAN 提出了一种算法来实现稳定的训练和高质量的输出。ACGAN 令人信服地提高了 MNIST 数字的条件生成质量，要求判别器在确定输入图像是假的还是真的基础上执行分类任务。在下一章，我们将研究如何控制生成器输出的

属性。虽然 CGAN 和 ACGAN 能够指出需要产生的数字，但我们还没有分析过能够指定输出属性的 GANs。例如，我们可能想控制 MNIST 数字的书写风格，如圆度、倾斜角度和厚度。因此，我们的目标将是引入具有分解表示的 GAN，以控制生成器输出的具体属性。

8.5. 参考文献

1. Ian Goodfellow et al.: Generative Adversarial Nets. Advances in neural information processing systems, 2014

2. Martin Arjovsky, Soumith Chintala, and Léon Bottou: Wasserstein GAN. arXiv preprint, 2017

3. Xudong Mao et al.: Least Squares Generative Adversarial Networks. 2017 IEEE International Conference on Computer Vision (ICCV). IEEE 2017

4. Augustus Odena, Christopher Olah, and Jonathon Shlens. Conditional Image Synthesis with Auxiliary Classifier GANs. ICML, 2017