

9

改进 GANs

自 2014 年引入生成对抗网络 (GANs) 以来 [1], 其受欢迎程度迅速提高。事实证明, GANs 是一种有用的生成模型, 可以合成看起来真实的新数据。随后的许多深度学习的研究论文提出了解决原始 GAN 的困难和限制的措施。

正如我们在前几章所讨论的, GANs 可能是出了名的难以训练, 而且容易出现模式崩溃。模式崩溃是指即使损失函数已经被优化, 但生成器产生的输出看起来还是一样的情况。在 MNIST 数字的背景下, 由于模式崩溃, 生成器可能只生成数字 4 和 9, 因为它们看起来很相似。Wasserstein GAN (WGAN) [2] 解决了这些问题, 它认为稳定的训练和模式崩溃可以通过简单地替换基于 Wasserstein 1 的 GAN 损失函数, 也被称为 Earth Mover's Distance (EMD) 来避免。

然而, 稳定性问题并不是 GANs 的唯一问题。提高生成图像的感知质量的需求也在不断增加。最小二乘法 GAN (LSGAN) [3] 建议同时解决这两个问题。其基本前提是西格玛交叉熵损失会导致训练期间梯度消失。这导致图像质量差。最小二乘法损失不会引起梯度消失。与 vanilla GAN 生成的图像相比, 所生成的图像具有更高的感知质量。

在上一章中, CGAN 介绍了一种调节生成器输出的方法。例如, 如果我们想得到数字 8, 我们会在生成器的输入中加入调节标签。受 CGAN 的启发, 辅助分类器 GAN (ACGAN) [4] 提出了一种改良的条件算法, 使输出的感知质量和多样性更好。

综上所述, 本章的目标是提出 - WGAN 的理论表述 - 对 LSGAN 原理的理解 - 对 ACGAN 原理的理解 - 改进的 GAN 的 tf.keras 实现 - WGAN、LSGAN 和 ACGAN

先来讨论一下 WGAN。

9.1. Wasserstein GAN

正如我们之前提到的, GANs 是出了名的难训练。鉴别器和生成器这两个网络的目标相反, 很容易导致训练不稳定。鉴别器试图从真实数据中对假数据进行正确分类。同时, 生成器尽力欺骗鉴别器。如果鉴别器的学习速度比生成器快, 生成器的参数将无法优化。另一方面, 如果鉴别器的学习速度较慢, 那么梯度可能在到达生成器之前就消失了。在最坏的情况下, 如果鉴别器无法收敛, 生成器将无法获得任何有用的反馈。WGAN 认为, GAN 的内在不稳定性是由于其损失函数, 它是

基于 Jensen-Shannon (JS) 距离的。在 GAN 中, 生成器的目标是学习如何从一个源分布 (例如, 噪声) 转换到一个估计的目标分布 (例如, MNIST 数字)。使用 GAN 的原始表述, 损失函数实际是最小化目标分布和其估计之间的距离。问题是, 对于某些分布对来说, 没有平稳的路径来最小化这个 JS 距离。因此, 训练将无法收敛。

在下一节中, 我们将研究三种距离函数, 并分析什么可以很好地替代 JS 距离函数, 更适合 GAN 优化。

9.1.1. 距离函数

训练 GAN 的稳定性可以通过检查其损失函数来了解。为了更好地理解 GAN 的损失函数, 我们将审查两个概率分布之间的常见距离或分歧函数。

我们关注的是真实数据分布的 p_{data} 和生成器数据分布的 p_g 之间的距离。GAN 的目标是使 $p_g \rightarrow p_{data}$ 。下列公式??, ??, ??, 显示了分歧函数。

在大多数最大似然任务中, 我们会在损失函数中使用 Kullback-Leibler(KL) 发散, 或 DKL, 作为衡量我们的神经网络模型预测与真实分布函数的距离。如公式??所示, DKL 不是对称的, 因为 $D_{KL}(P_{fsts}||p_g) \neq D_{KL}(p_g||p_{data})$ 。

JS, 即 D_{JS} , 是一种基于 D_{KL} 的分歧。然而, 与 D_{KL} 不同, D_{JS} 是对称的, 并且是有限的。在本节中, 我们将证明优化 GAN 损失函数等同于优化 D_{JS} 。

Kullback- Leibler (KL) :

$$E : 8 - 1D_{KL}(p_{data}||p_g) = \mathbb{E}_{x \sim p_{data}} \log \frac{p_{data}(x)}{p_g(x)} \quad (p_g||p_{data}) = \mathbb{E}_{x \sim p_g} \log \frac{p_g(x)}{p_{data}(x)} \quad (9.1)$$

Jensen- Shannon (JS) :

$$E : 8 - 2D_{JS}(P_{data}||p_g) = \frac{1}{2} \mathbb{E}_{x \sim p_{data}} \log \frac{p_{data}(x)}{\frac{p_{data}(x)+p_g(x)}{2}} + \frac{1}{2} \mathbb{E}_{x \sim p_g} \log \frac{p_g(x)}{\frac{p_{data}(x)+p_g(x)}{2}} = D_{JS}(p_g||p_{data}) \quad (9.2)$$

EMD 的理念是, 它是衡量概率分布 p_{data} 的质量 $\gamma(x, y)$ 应该由 $d = \|x - y\|$ 是所有可能的联合分布 $\prod(p_{data}, p_g)$ 空间中的联合分布。 $\gamma(x, y)$ 也被称为运输计划, 以反映运输质量以匹配两个概率分布的策略。鉴于两个概率分布, 有许多可能的运输计划。粗略地说, \inf 表示一个成本最低的运输计划。

例如, 图9.1向我们展示了两个简单的离散分布 x 和 y 。

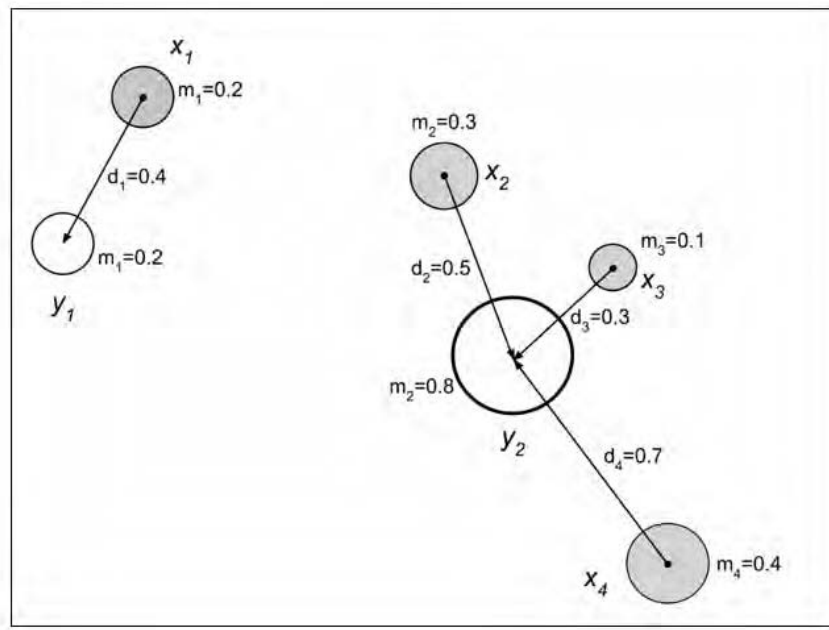


图 9.1: EMD 是为了与目标分布相匹配而从 x 中运输的质量的加权量, y

x 在 $i = 1, 2, 3$ 和 4 的位置 x_i 有质量 m_i , $i = 1, 2, 3$ 和 4 。同时, y 在 $i = 1$ 和 2 的位置 y_i 有质量 m_i , $i = 1$ 和 2 。为了与分布 y 相匹配, 箭头显示了用 d_i 移动每个质量 x_i 的最小运输计划。EMD 的计算方法是:

$$EMD = \sum_{i=1}^4 x_i d_i = 0.1(0.4) + 0.3(0.5) + 0.1(0.3) + 0.4(0.7) = 0.54 \quad (9.3)$$

在图9.1中, EMD 可以解释为移动土堆 x 以填满孔洞 y 所需的最小工作量。虽然在这个例子中, \inf 也可以从图中推导出来, 但在大多数情况下, 特别是在连续分布中, 要穷尽所有可能的运输计划是难以做到的。我们将在本章的后面再来讨论这个问题。同时, 我们将展示 GAN 的损失函数实际上是如何使 JS 发散最小化的。

9.1.2. GANs 中的距离函数

我们现在要计算的是, 从上一章的损失函数中给定任何生成器的最佳判别器。我们回顾一下上一章中的以下方程式。

$$L^D = -\mathbb{E}_{x \sim p_{data}} \log D(x) - \mathbb{E} \log(1 - D(g(z))) \quad (9.4)$$

代替从噪声分布中取样, 前述方程也可以表示为从发生器分布中取样:

$$L^D = -\mathbb{E}_{x \sim p_{data}} \log D(x) - \mathbb{E} \log(1 - D(x)) \quad (9.5)$$

要找到最小的 L^D 。

$$L^D = -\int_x p_{data}(x) \log D(x) dx - \int_x p_g(x) \log(1 - D(x)) dx \quad (9.6)$$

$$L^D = - \int_x p_{data}(x) \log D(x) dx + p_g(x) \log(1 - D(x)) dx \quad (9.7)$$

积分内的项的形式是: $y \rightarrow a \log y + b \log(1 - y)$, 对于 $y \in [0, 1]$ 的任何 $\frac{a}{a+b}$, $b \in \mathbb{R}^2$ 不包括 0, 它有一个已知最大值。由于积分不会改变这个表达式的最大值 (或 L^D 的最小值) 的位置, 所以最优判别器是:

$$D^*(x) = \frac{p_{data}}{p_{data} + p_g} \quad (9.8)$$

因此, 给定最优判别器的损失函数为:

$$L(D^*) = -\mathbb{E}_{x \sim p_{data}} \log \frac{p_{data}}{p_{data} + p_g} - \mathbb{E}_{x \sim p_g} \log(1 - \frac{p_{data}}{p_{data} + p_g}) \quad (9.9)$$

$$L(D^*) = -\mathbb{E}_{x \sim p_{data}} \log \frac{p_{data}}{p_{data} + p_g} - \mathbb{E}_{x \sim p_g} \log(\frac{p_g}{p_{data} + p_g})$$

$$L(D^*) = -\mathbb{E}_{x \sim p_{data}} \log \frac{p_{data}}{p_{data} + p_g} - \mathbb{E}_{x \sim p_g} \log(\frac{p_g}{p_{data} + p_g}) \quad (9.11)$$

$$L(D^*) = 2 \log 2 - D_{KL}(P_{data} || \frac{p_{data} + p_g}{2}) - D_{KL}(p_g || \frac{p_{data} + p_g}{2}) \quad (9.12)$$

$$E : 5.1.12 L(D^*) = 2 \log 2 - D_{js}(p_{data} || p_g) \quad (9.13)$$

从方程??可以看出, 最优判别器的损失函数是一个常数减去真实分布 p_{data} 和任何生成器分布 p_g 之间的 JS 分歧的两倍。最小化 $L(D^*)$ 意味着最大化 $D_{JS}(p_{data} || p_g)$, 否则判别器必须从真实数据中正确分类。

同时, 我们可以有把握地认为, 当生成器的分布等于真实数据的分布时, 就是最佳生成器。

$$g^*(x) \rightarrow p_g = p_{data} \quad (9.14)$$

这是有道理的, 因为生成器的目标是通过学习真实的数据分布来欺骗判别器。实际上, 我们可以通过最小化 D_{JS} 或使 $p_g \rightarrow p_{data}$, 来达到最佳生成器。给定一个最优生成器, 最优判别器是 $D^*(x) = \frac{1}{2}$ 其中 $L^{D^*} = 2 \log 2 = 0.60$ 。

问题是, 当两个分布没有重叠的时候, 没有任何平滑函数可以帮助缩小它们之间的差距。训练 GANs 不会通过梯度下降收敛。例如:

$$p_{data} = (x, y) \text{ where } x = 0, y \sim U(0, 1) \quad (9.15)$$

$$p_g = (x, y) \text{ where } x = 1, y \sim U(0, 1) \quad (9.16)$$

这两个分布在图9.2中显示。

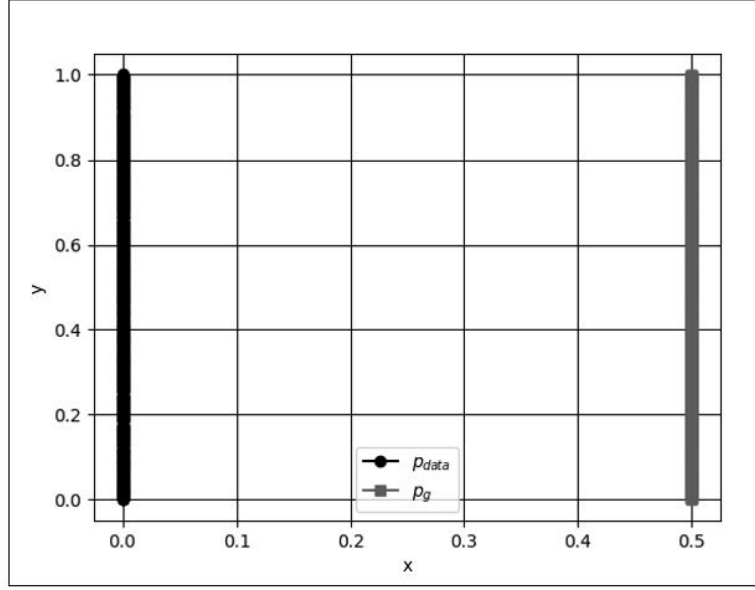


图 9.2: 两个没有重叠的分布的例子。 $\Theta = 0.5$ 及 p_g

$U(0, 1)$ 是均匀分布。各个距离函数的发散情况如下：

$$\begin{aligned} D_{KL}(p_{data}||p_g) &= \mathbb{E}_{x=0,y \sim U(0,1)} \log \frac{p_{data}(x,y)}{p_g(x,y)} \\ &= \sum 1 \log \frac{1}{0} \\ &= +\infty \end{aligned} \quad (9.17)$$

$$\begin{aligned} D_{KL}(p_g||p_{data}) &= \mathbb{E}_{x=0,y \sim U(0,1)} \log \frac{p_g(x,y)}{p_{data}(x,y)} \\ &= \sum 1 \log \frac{1}{0} \\ &= +\infty \end{aligned} \quad (9.18)$$

$$\begin{aligned} D_{JS}(p_{data}||p_g) &= \frac{1}{2} \mathbb{E}_{x=0,y \sim U(0,1)} \log \frac{p_{data}(x,y) + p_g(x,y)}{2} + \\ \frac{1}{2} \mathbb{E}_{x=0,y \sim U(0,1)} \log \frac{p_g(x,y) + p_{data}(x,y)}{2} &= \frac{1}{2} \sum 1 \log \frac{1}{\frac{1}{2}} + \frac{1}{2} \sum 1 \log \frac{1}{\frac{1}{2}} \\ &= \log 2 \end{aligned} \quad (9.19)$$

$$W(p_{data}, p_g) = |\Theta| \quad (9.20)$$

由于 D_{JS} 是一个常数，GAN 不会有足够的梯度来驱动 $p_g \rightarrow p_{data}$ 。我们也会发现， D_{KL} 或反向 D_{KL} ，也是没有用的。然而，对于 $W(p_{data}, p_g)$ ，我们可以有一个平滑函数，以便通过梯度下降达到 $p_g \rightarrow p_{data}$ 。EMD 或 Wasserstein 1 似乎是一个更合理的损失函数，以优化 GANs，因为

D_{JS} 在两个分布有最小或没有重叠的情况下会失败。¹

在下一节中，我们将专注于使用 **EMD** 或 **Wasserstein 1** 距离函数来开发一个替代损失函数，以鼓励 **GANs** 的稳定训练。

9.1.3. 使用 Wasserstein 损失函数

在使用 **EMD** 或 **Wasserstein 1** 之前，还有一个问题需要克服。要穷尽 $\Pi(p_{data}, p_g)$ 的空间来寻找 $\gamma \in \Pi(p_{data}, p_g)$ 是难以做到的。建议的解决方案是使用其康托罗维奇-鲁宾斯坦 (**Kantorovich-Rubinstein**) 对偶。

$$W(p_{data}, p_g) = \frac{1}{K} \sup \mathbb{E}_x p_{data} f(x) - \mathbb{E}_x p_g f(x)$$

等价地 **EMD** 是所有 **K-Lipschitz** 函数的最高值（约是最大值）。**K-Lipschitz** $f: X \rightarrow \mathbb{R}$ 函数满足约束条件。

$$|f(x_1) - f(x_2)| \leq K|x_1 - x_2| \quad (9.22)$$

对于所有 $x_1, x_2 \in \mathbb{R}$ 。**K-Lipschitz** 函数具有有界导数，并且几乎总是连续可微的（例如， $f(x) = |x|$ 具有有界导数，并且在 $x = 0$ 处连续但不可微）。

方程??可以通过寻找 **K-Lipschitz** 函数族 $\{f_w\}$ 解决。

$$W(p_{data}, p_g) = \max \mathbb{E}_x p_{data} [f_w(x)] - \mathbb{E}_x p_g [f_w(x)] \quad (9.23)$$

在 **GANs** 的背景下，方程9.23可以通过从 **z-noise** 分布中取样并以判别函数 D_w 代替 f_w 来重写。

$$W(p_{data}, p_g) = \max \mathbb{E}_x p_{data} D_w(x) - \mathbb{E}_x p_g D_w(g(z))$$

最后一个问题是如何找到函数族， $w \in W$ 。建议的解决方案是在每次梯度更新时；判别器 w 的权重在下限和上限之间进行剪裁（例如，-0.01 和 0.01）。

$$w \leftarrow \text{clip}(w, -0.01, 0.01) \quad (9.25)$$

w 的小值将鉴别器约束在一个紧凑的参数空间内，从而确保 **Lipschitz** 的连续性。我们可以使用方程??作为我们新的 **GAN** 损失函数的基础。**EMD** 或 **Wasserstein 1** 是生成器旨在最小化的损失函数，也是判别器试图最大化（或最小化 $-W(p_{data}, p_g)$ ）的成本函数。

$$L^D = -\mathbb{E}_x p_{data} D_w(x) + \mathbb{E}_z D_w(g(z)) \quad (9.26)$$

$$L^G = -\mathbb{E}_z D_w(g(z)) \quad (9.27)$$

¹为了帮助进一步理解，关于距离函数的出色讨论可以在以下网站找到：<https://lilianweng.github.io/lil-log/2017/08/20/from-GAN-to-WGAN.html>。

在生成器的损失函数中，第一个项消失了，因为它不是直接针对真实数据进行优化。

公式9.28至公式9.32显示了 GAN 和 WGAN 的损失函数之间的区别。为了简洁起见，简化了 L^D 和 L^G 的符号。

GAN:

$$L^D = -\mathbb{E}_{x \sim p_{data}} \log D(x) - \mathbb{E}_z \log(1 - D_w(g(z))) \quad (9.28)$$

$$L^G = -\mathbb{E}_z \log D(g(z)) \quad (9.29)$$

WGAN:

$$L^D = -\mathbb{E}_{x \sim p_{data}} D_w(x) + \mathbb{E}_z D_w(g(z)) \quad (9.30)$$

$$L^G = -\mathbb{E}_z D_w(g(z)) \quad (9.31)$$

$$w \leftarrow \text{clip}(w, -0.01, 0.01) \quad (9.32)$$

这些损失函数被用于训练 WGAN，如下列算法所示。参数值为 $a = 0.00005, c = 0.01, m = 64, n_{critic} = 5$ 。其中： a : 学习率。 c : 剪裁参数。 m : 批次大小。 n_{critic} : 每个发生器迭代的判别器迭代次数。其中： w_n : 初始批判器（判别器）参数。 Θ_n : 初始生成器参数。

1. while Θ has not converged do
2. for $t = 1, \dots, n_{critic}$ do
3. Sample a batch $\{x^{(i)}\}_{i=1}^m \sim p_{data}$ from real data
4. Sample a batch $\{z^{(i)}\}_{i=1}^m \sim p_z$ from uniform noise distribution
5. $g_w \leftarrow \nabla_w - \frac{1}{m} \sum_{i=1}^m D_w(x^{(i)}) + \frac{1}{m} \sum_{i=1}^m D_w(z^{(i)})$, compute discriminator gradients
6. $w \leftarrow w - \alpha \times \text{RMSPProp}(w, g_w)$, update discriminator parameters
7. $w \leftarrow \text{clip}(w, -c, c)$, clip discriminator weights
8. end for
9. Sample a batch $\{z^{(i)}\}_{i=1}^m \sim p_z$ 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. $\Theta \leftarrow \Theta - \alpha \times \text{RMSPProp}(\Theta, g_\Theta)$, 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

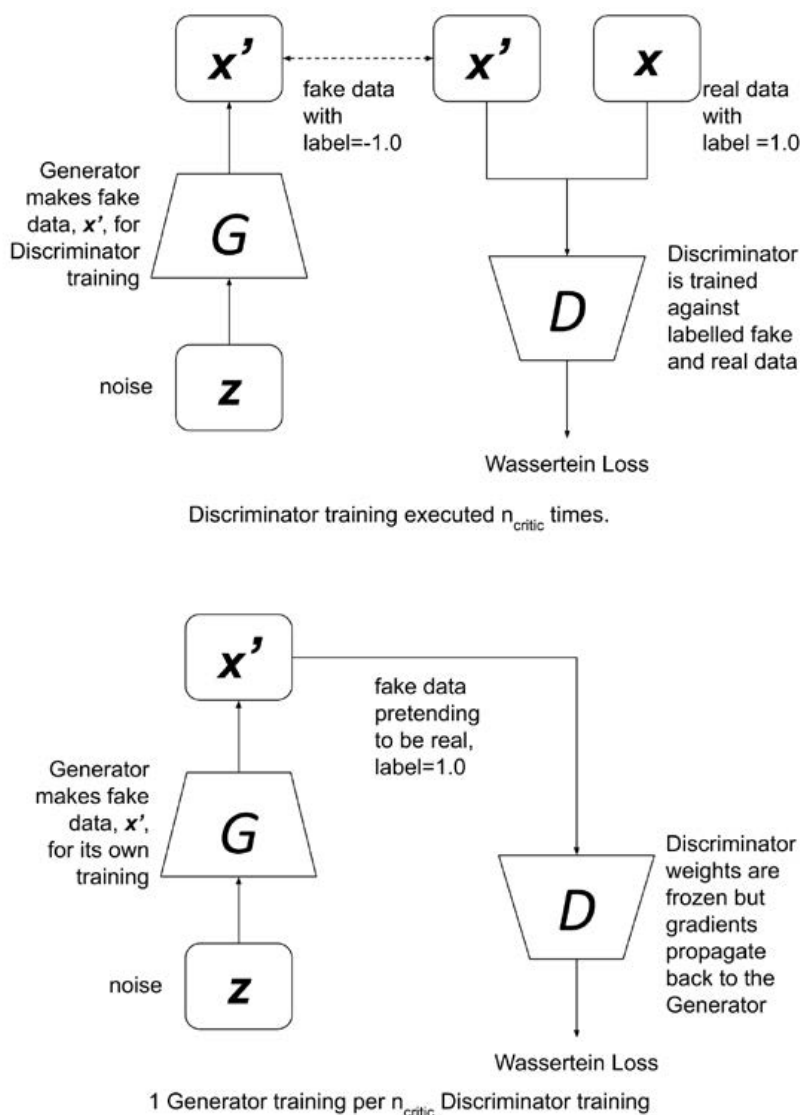


图 9.3: 上半部分: 训练 WGAN 判别器需要来自生成器的假数据和来自真实分布的真实数据。下半部分: 训练 WGAN 生成器需要来自生成器的假数据, 假装是真实的。

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 行计算生成器梯度，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：

$$L = -y_{label} \frac{1}{m} \sum_{i=1}^m y_{pred} \quad (9.33)$$

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 \square 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。

9.1.4. 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 9.1: RMSprop 优化器和 Wasserstein 损失函数的使用

```
1 Listing 5.1.1: wgan-mnist-5.1.2.py
2 def build_and_train_models():
3     """Load the dataset, build WGAN discriminator,
4     generator, and adversarial models.
5     Call the WGAN train routine.
6     """
7     # load MNIST dataset
8     (x_train, _), (_, _) = mnist.load_data()
9     # reshape data for CNN as (28, 28, 1) and normalize
```

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

```

10     image_size = x_train.shape[1]
11     x_train = np.reshape(x_train, [-1, image_size, image_size, 1])
12     x_train = x_train.astype('float32') / 255
13     model_name = "wgan_mnist"
14 # network parameters
15 # the latent or z vector is 100-dim latent_size = 100
16 # hyper parameters from WGAN paper [2] n_critic = 5
17 clip_value = 0.01
18 batch_size = 64
19 lr = 5e-5
20 train_steps = 40000
21 input_shape = (image_size, image_size, 1)
22 # build discriminator model
23 inputs = Input(shape=input_shape, name='discriminator_input') # WGAN uses linear activation in
    paper [2]
24 discriminator = gan.discriminator(inputs, activation='linear') optimizer = RMSprop(lr=lr)
25 # WGAN discriminator uses wasserstein loss discriminator.compile(loss=wasserstein_loss,
    optimizer=optimizer,
26     metrics=['accuracy'])
27 discriminator.summary()
28 # build generator model
29 input_shape = (latent_size, )
30 inputs = Input(shape=input_shape, name='z_input')
31 generator = gan.generator(inputs, image_size)
32 generator.summary()
33 # build adversarial model = generator + discriminator
34 # freeze the weights of discriminator during adversarial training
35 discriminator.trainable = False
36 adversarial = Model(inputs,
    discriminator(generator(inputs)),
37     name=model_name)
38 adversarial.compile(loss=wasserstein_loss,
    optimizer=optimizer,
39     metrics=['accuracy'])
40 adversarial.summary()
41 # train discriminator and adversarial networks
42 models = (generator, discriminator, adversarial)
43 params = (batch_size,
    latent_size,
44     n_critic,
45     clip_value,
46     train_steps,
47     model_name)
48 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 9.2: Evaluate how the model does on the test set

```

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

```

```

4 Discriminator is trained first with properly labelled real and fake images for n_critic times.
   Discriminator weights are clipped as a requirement
5 of Lipschitz constraint.
6 Generator is trained next (via Adversarial) with
7 fake images pretending to be real.
8 Generate sample images per save_interval
9 Arguments:
10     models (list): Generator, Discriminator,
11         Adversarial models
12     x_train (tensor): Train images
13     params (list) : Networks parameters
14 """
15 # the GAN models
16 generator, discriminator, adversarial = models
17 # network parameters
18 (batch_size, latent_size, n_critic,
19     clip_value, train_steps, model_name) = params
20 # the generator image is saved every 500 steps
21 save_interval = 500
22 # noise vector to see how the
23 # generator output evolves during training
24 noise_input = np.random.uniform(-1.0,
25     1.0,
26         size=[16, latent_size])
27 # number of elements in train dataset
28 train_size = x_train.shape[0]
29 # labels for real data
30 real_labels = np.ones((batch_size, 1))
31 for i in range(train_steps):
32     # train discriminator n_critic times
33     loss = 0
34     acc = 0
35     for _ in range(n_critic):
36         # train the discriminator for 1 batch
37         # 1 batch of real (label=1.0) and
38         # fake images (label=-1.0)
39         # randomly pick real images from dataset
40         rand_indexes = np.random.randint(0,
41 train_size,
42         size=batch_size)
43         real_images = x_train[rand_indexes]
44         # generate fake images from noise using generator
45         # generate noise using uniform distribution
46         noise = np.random.uniform(-1.0,
47     1.0,
48         size=[batch_size, latent_size])
49         fake_images = generator.predict(noise)
50         # train the discriminator network
51         # real data label=1, fake data label=-1
52         constraint
53 # instead of 1 combined batch of real and fake images,
54 # train with 1 batch of real data first, then 1 batch
55 # of fake images.
56 # this tweak prevents the gradient
57 # from vanishing due to opposite
58 # signs of real and fake data labels (i.e. +1 and -1) and
59 # small magnitude of weights due to clipping.

```

```

60 real_loss, real_acc = \
61     discriminator.train_on_batch(real_images,
62                                 real_labels)
63 fake_loss, fake_acc = \
64     discriminator.train_on_batch(fake_images,
65                                 -real_labels)
66 # accumulate average loss and accuracy
67 loss += 0.5 * (real_loss + fake_loss)
68 acc += 0.5 * (real_acc + fake_acc)
69 # clip discriminator weights to satisfy Lipschitz
70 for layer in discriminator.layers:
71     weights = layer.get_weights()
72     weights = [np.clip(weight,
73                         -clip_value,
74                         clip_value) for weight in weights]
75     layer.set_weights(weights)
76 # average loss and accuracy per n_critic training iterations
77 loss /= n_critic
78 acc /= n_critic
79 log = "%d: [discriminator loss: %f, acc: %f]" % (i, loss, acc)
80 # train the adversarial network for 1 batch
81 # 1 batch of fake images with label=1.0
82 # since the discriminator weights are frozen in
83 # adversarial network only the generator is trained
84 # generate noise using uniform distribution
85 noise = np.random.uniform(-1.0,
86                             1.0,
87                             size=[batch_size, latent_size])
88 # train the adversarial network
89 # note that unlike in discriminator training,
90 # we do not save the fake images in a variable
91 # the fake images go to the discriminator
92 # input of the adversarial for classification
93 # fake images are labelled as real
94 # log the loss and accuracy
95 loss, acc = adversarial.train_on_batch(noise, real_labels)
96 log = "%s [adversarial loss: %f, acc: %f]" % (log, loss, acc)
97 print(log)
98 if (i + 1) % save_interval == 0:
99     # plot generator images on a periodic basis
100     gan.plot_images(generator,
101                     noise_input=noise_input,
102                     show=False,
103                     step=(i + 1),
104                     model_name=model_name)
105     # save the model after training the generator
106     # the trained generator can be reloaded
107     # for future MNIST digit generation
108     generator.save(model_name + ".h5")

```

图9.4显示了 WGAN 在 MNIST 数据集上输出的演变。

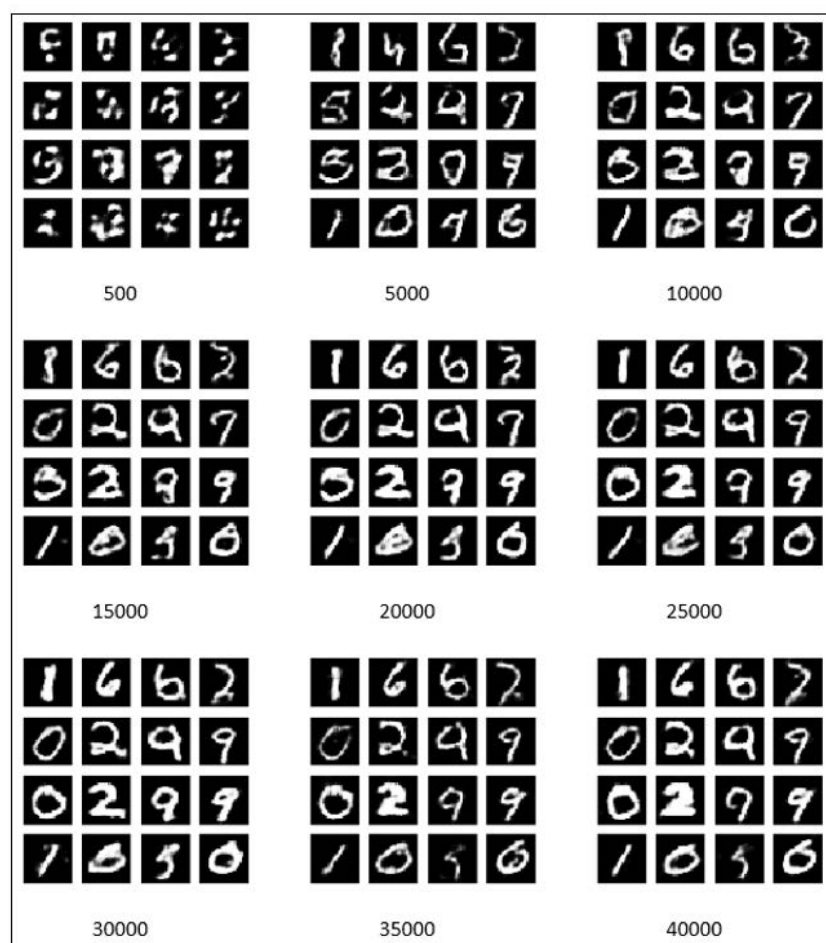


图 9.4: WGAN 的样本输出与训练步骤的对比。在训练和测试期间, WGAN 的任何输出都没有出现模式崩溃。

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。

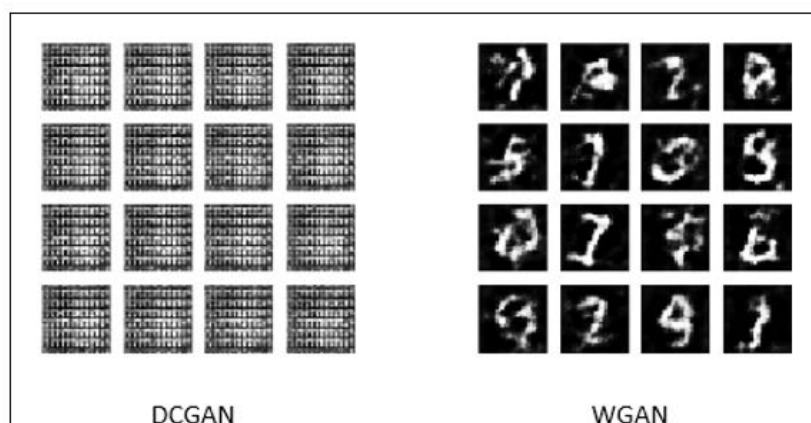


图 9.5: 在判别器网络的 ReLU 激活之前插入批量标准化时, DCGAN (左) 和 WGAN (右) 的输出比较

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。

9.2. 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。

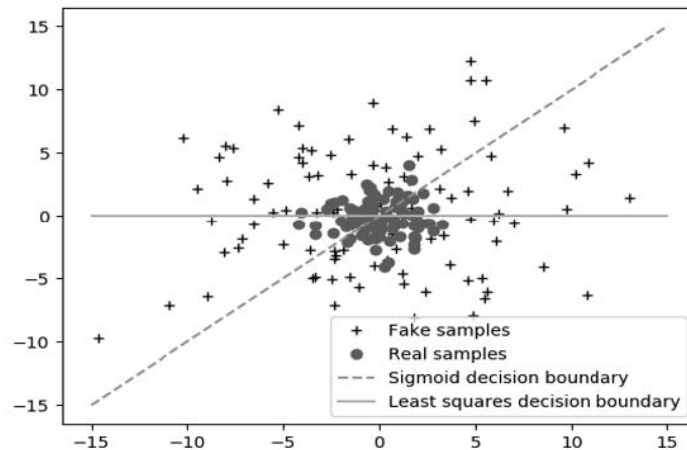


图 9.6: Both real and fake sample distributions divided by their respective decision boundaries: sigmoid and least squares

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 9.3: 建立鉴别器

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

该发生器与原来的 DCGAN 类似。

Listing 9.4: 建立发生器

```
1 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 9.5: Evaluate how the model does on the test set

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

Listing 9.6: Evaluate how the model does on the test set

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

```

2   def build_and_train_models():
3       """Load the dataset, build LSGAN discriminator,
4       generator, and adversarial models.
5       Call the LSGAN train routine.
6       """
7       # load MNIST dataset
8       (x_train, _), (_, _) = mnist.load_data()
9       # reshape data for CNN as (28, 28, 1) and normalize
10      image_size = x_train.shape[1]
11      x_train = np.reshape(x_train,
12                          [-1, image_size, image_size, 1])
13      x_train = x_train.astype('float32') / 255
14      model_name = "lsGAN_mnist"
15      # network parameters
16      # the latent or z vector is 100-dim
17      latent_size = 100
18      input_shape = (image_size, image_size, 1)
19      batch_size = 64
20      lr = 2e-4
21      decay = 6e-8
22      train_steps = 40000
23      # build discriminator model
24      inputs = Input(shape=input_shape, name='discriminator_input') discriminator = gan.discriminator(
25          inputs, activation=None)
26      # [1] uses Adam, but discriminator easily
27      # converges with RMSprop
28      optimizer = RMSprop(lr=lr, decay=decay)
29      # LSGAN uses MSE loss [2]
30      discriminator.compile(loss='mse',
31                          optimizer=optimizer,
32                          metrics=['accuracy'])
33      discriminator.summary()
34      # build generator model
35      input_shape = (latent_size, )
36      inputs = Input(shape=input_shape, name='z_input')
37      generator = gan.generator(inputs, image_size)
38      generator.summary()
39      # build adversarial model = generator + discriminator
40      optimizer = RMSprop(lr=lr*0.5, decay=decay*0.5)
41      # freeze the weights of discriminator
42      # during adversarial training
43      discriminator.trainable = False
44      adversarial = Model(inputs,
45                          discriminator(generator(inputs)),
46                          name=model_name)
47      # LSGAN uses MSE loss [2]
48      adversarial.compile(loss='mse',
49                          optimizer=optimizer,
50                          metrics=['accuracy'])
51      adversarial.summary()
52      # train discriminator and adversarial networks
53      models = (generator, discriminator, adversarial)
54      params = (batch_size, latent_size, train_steps, model_name)
55      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。

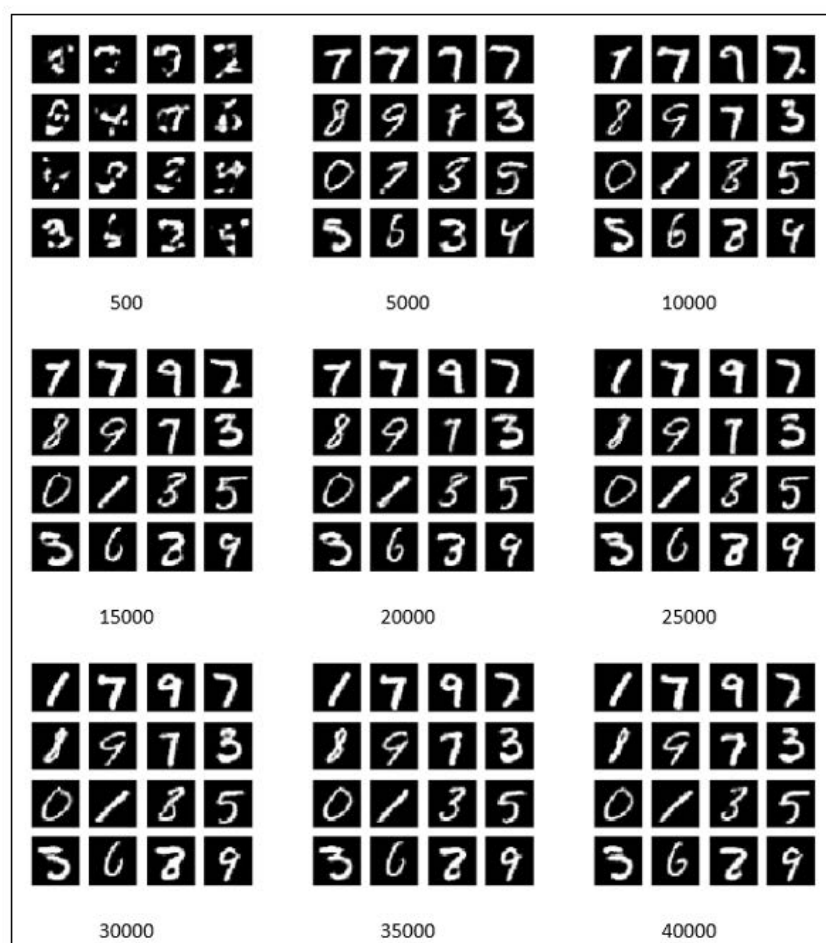


图 9.7: LSGAN 的样本输出与训练步骤的对比

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 9.7: Evaluate how the model does on the test set

```
1 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。

9.3. 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。

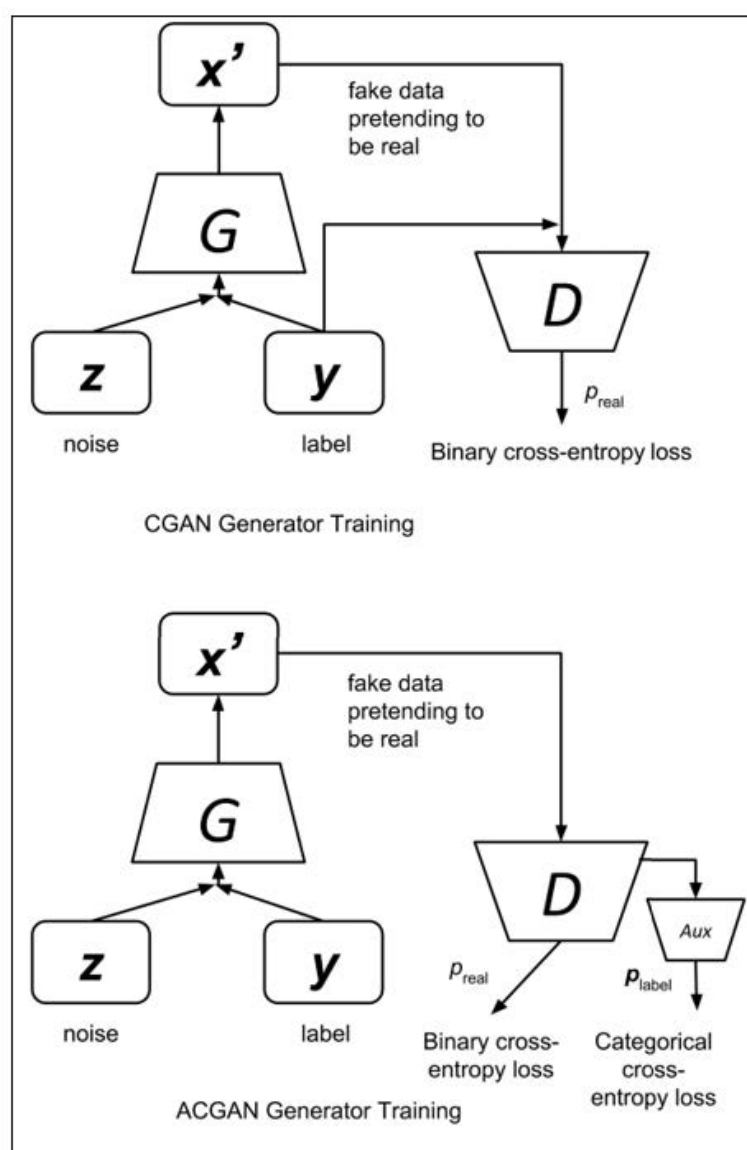


图 9.8: CGAN 与 ACGAN 发生器的训练。主要的区别是判别器的输入和输出

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 ($-\mathbb{E}_{x \sim p_{\text{data}}} \log D(x|y) - \mathbb{E}_z \log(1 - D(g(z|y)))$)，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 ($-\mathbb{E}_{x \sim p_{\text{data}}} \log p(c|x) - \mathbb{E}_z \log(c|g(z|y))$)。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。鉴别器和生成器构造函数也由 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 9.8: Listing 5.3.1: gan.py

```

1 Listing 5.3.1: gan.py
2 def discriminator(inputs,
3                 activation='sigmoid',
4                 num_labels=None,
5                 num_codes=None):
6     """Build a Discriminator Model
7     Stack of LeakyReLU-Conv2D to discriminate real from fake
8     The network does not converge with BN so it is not used here
9     unlike in [1]
10    Arguments:
11    inputs (Layer): Input layer of the discriminator (the image) activation (string): Name of output
12    activation layer num_labels (int): Dimension of one-hot labels for ACGAN &
13    InfoGAN
14    num_codes (int): num_codes-dim Q network as output
15    if StackedGAN or 2 Q networks if InfoGAN
16    Returns:
17    Model: Discriminator Model
18    """
19    kernel_size = 5
20    layer_filters = [32, 64, 128, 256]
21    x = inputs
22    for filters in layer_filters:
23        # first 3 convolution layers use strides = 2
24        # last one uses strides = 1
25        if filters == layer_filters[-1]:
26            strides = 1
27        else:
28            strides = 2
29        x = LeakyReLU(alpha=0.2)(x)
30        x = Conv2D(filters=filters,
31                  kernel_size=kernel_size,
32                  strides=strides,
33                  padding='same')(x)
34    x = Flatten()(x)
35    # default output is probability that the image is real
36    outputs = Dense(1)(x)
37    if activation is not None:
38        print(activation)
39        outputs = Activation(activation)(outputs)
40    if num_labels:
41        # ACGAN and InfoGAN have 2nd output
42        # 2nd output is 10-dim one-hot vector of label
43        layer = Dense(layer_filters[-2])(x)
44        labels = Dense(num_labels)(layer)
45        labels = Activation('softmax', name='label')(labels)
46        if num_codes is None:
47            outputs = [outputs, labels]
48        else:
49            # InfoGAN have 3rd and 4th outputs
50            # 3rd output is 1-dim continuous Q of 1st c given x

```

```

50         code1 = Dense(1)(layer)
51         code1 = Activation('sigmoid', name='code1')(code1)
52         # 4th output is 1-dim continuous Q of 2nd c given x
53         code2 = Dense(1)(layer)
54         code2 = Activation('sigmoid', name='code2')(code2)
55         outputs = [outputs, labels, code1, code2]
56     elif num_codes is not None:
57         # StackedGAN Q0 output
58         # z0_recon is reconstruction of z0 normal distribution
59         z0_recon = Dense(num_codes)(x)
60         z0_recon = Activation('tanh', name='z0')(z0_recon)
61         outputs = [outputs, z0_recon]
62     return Model(inputs, outputs, name='discriminator')

```

然后通过调用建立判别器：

Listing 9.9: Evaluate how the model does on the test set

```

1 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 9.10: Listing 5.3.2: gan.py

```

1 Listing 5.3.2: gan.py
2     def generator(inputs,
3         image_size,
4         activation='sigmoid',
5         labels=None,
6         codes=None):
7         """Build a Generator Model
8         Stack of BN-ReLU-Conv2DTranpose to generate fake images.
9         Output activation is sigmoid instead of tanh in [1].
10        Sigmoid converges easily.
11        Arguments:
12        inputs (Layer): Input layer of the generator (the z-vector)
13        image_size (int): Target size of one side
14            (assuming square image)
15        activation (string): Name of output activation layer
16        labels (tensor): Input labels
17        codes (list): 2-dim disentangled codes for InfoGAN
18    Returns:
19        Model: Generator Model
20    """
21    image_resize = image_size // 4
22    # network parameters
23    kernel_size = 5
24    layer_filters = [128, 64, 32, 1]
25    if labels is not None:
26        if codes is None:
27            # ACGAN labels
28            # concatenate z noise vector and one-hot labels
29            inputs = [inputs, labels]
30        else:
31            # infoGAN codes

```

```

32     # concatenate z noise vector,
33     # one-hot labels and codes 1 & 2
34     inputs = [inputs, labels] + codes
35     x = concatenate(inputs, axis=1)
36 elif codes is not None:
37     # generator 0 of StackedGAN
38     inputs = [inputs, codes]
39     x = concatenate(inputs, axis=1)
40 else:
41     # default input is just 100-dim noise (z-code)
42     x = inputs
43 x = Dense(image_resize * image_resize * layer_filters[0])(x)
44 x = Reshape((image_resize, image_resize, layer_filters[0]))(x)
45 for filters in layer_filters:
46     # first two convolution layers use strides = 2
47     # the last two use strides = 1
48     if filters > layer_filters[-2]:
49         strides = 2
50     else:
51         strides = 1
52     x = BatchNormalization()(x)
53     x = Activation('relu')(x)
54     x = Conv2DTranspose(filters=filters,
55                        kernel_size=kernel_size,
56                        strides=strides,
57                        padding='same')(x)
58     if activation is not None:
59         x = Activation(activation)(x)
60     # generator output is the synthesized image x
61     return Model(inputs, x, name='generator')

```

在 ACGAN 中，发生器被实例化为：

Listing 9.11: 在 ACGAN 中，发生器被实例化程序

```

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

```

图、9.9显示了 `tf.keras` 中 ACGAN 的网络模型。

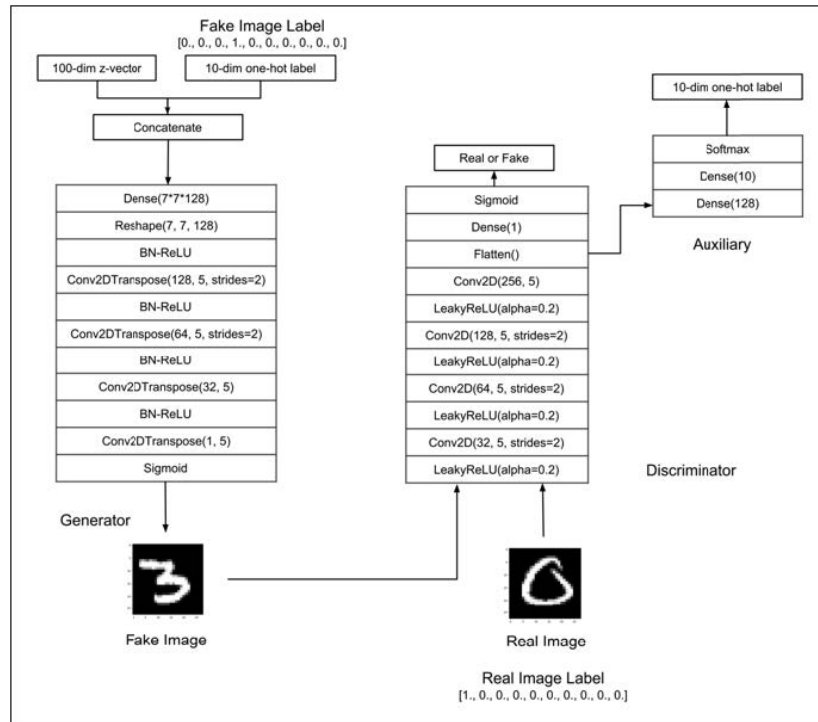


图 9.9: ACGAN 的 tf.keras 模型

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

```

1 def build_and_train_models():
2     """Load the dataset, build ACGAN discriminator,
3     generator, and adversarial models.
4     Call the ACGAN train routine.
5     """
6     # load MNIST dataset
7     (x_train, y_train), (_, _) = mnist.load_data()
8     # reshape data for CNN as (28, 28, 1) and normalize
9     image_size = x_train.shape[1]
10    x_train = np.reshape(x_train,
11                        [-1, image_size, image_size, 1])
12    x_train = x_train.astype('float32') / 255
13    # train labels
14    num_labels = len(np.unique(y_train))
15    y_train = to_categorical(y_train)
16    model_name = "acgan_mnist"
17    # network parameters
18    latent_size = 100
19    batch_size = 64
20    train_steps = 40000

```

```

21     lr = 2e-4
22     decay = 6e-8
23     input_shape = (image_size, image_size, 1)
24     label_shape = (num_labels, )
25     # build discriminator Model
26     inputs = Input(shape=input_shape,
27                     name='discriminator_input')
28     # call discriminator builder
29     # with 2 outputs, pred source and labels
30     discriminator = gan.discriminator(inputs,
31 num_labels=num_labels)
32 # [1] uses Adam, but discriminator
33 # easily converges with RMSprop
34     optimizer = RMSprop(lr=lr, decay=decay)
35     # 2 loss fuctions: 1) probability image is real
36     # 2) class label of the image
37     loss = ['binary_crossentropy', 'categorical_crossentropy']
38     discriminator.compile(loss=loss,
39                           optimizer=optimizer,
40                           metrics=['accuracy'])
41     discriminator.summary()
42 # build generator model
43 input_shape = (latent_size, )
44 inputs = Input(shape=input_shape, name='z_input') labels = Input(shape=label_shape, name='labels')
45     # call generator builder with input labels generator = gan.generator(inputs,
46                                     image_size,
47                                     labels=labels)
48     generator.summary()
49     # build adversarial model = generator + discriminator
50     optimizer = RMSprop(lr=lr*0.5, decay=decay*0.5)
51     # freeze the weights of discriminator
52     # during adversarial training
53     discriminator.trainable = False
54     adversarial = Model([inputs, labels],
55                         discriminator(generator([inputs, labels])),
56                         name=model_name)
57     # same 2 loss fuctions: 1) probability image is real
58     # 2) class label of the image
59     adversarial.compile(loss=loss,
60                        optimizer=optimizer,
61                        metrics=['accuracy'])
62     adversarial.summary()
63     # train discriminator and adversarial networks
64     models = (generator, discriminator, adversarial)
65     data = (x_train, y_train)
66     params = (batch_size, latent_size, \
67               train_steps, num_labels, model_name)
68     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 9.13: Listing 5.3.4: acgan-mnist-5.3.1.py

```

1 %Listing 5.3.4: acgan-mnist-5.3.1.py
2 def train(models, data, params):
3     """Train the discriminator and adversarial Networks
4     Alternately train discriminator and adversarial

```

```

5     networks by batch.
6     Discriminator is trained first with real and fake
7     images and corresponding one-hot labels.
8     Adversarial is trained next with fake images pretending
9     to be real and corresponding one-hot labels.
10    Generate sample images per save_interval.
11    # Arguments
12        models (list): Generator, Discriminator,
13            Adversarial models
14        data (list): x_train, y_train data
15        params (list): Network parameters
16    """
17    # the GAN models
18    generator, discriminator, adversarial = models
19    # images and their one-hot labels
20    x_train, y_train = data
21    # network parameters
22    batch_size, latent_size, train_steps, num_labels, model_name \
23        = params
24    # the generator image is saved every 500 steps
25    save_interval = 500
26    # noise vector to see how the generator
27    # output evolves during training
28    noise_input = np.random.uniform(-1.0,
29    1.0,
30                                size=[16, latent_size])
31    # class labels are 0, 1, 2, 3, 4, 5,
32    # 6, 7, 8, 9, 0, 1, 2, 3, 4, 5
33    # the generator must produce these MNIST digits
34    noise_label = np.eye(num_labels)[np.arange(0, 16) % num_labels] # number of elements in train
35    dataset
36    train_size = x_train.shape[0]
37    print(model_name,
38          "Labels for generated images: ",
39          np.argmax(noise_label, axis=1))
40    for i in range(train_steps):
41        # train the discriminator for 1 batch
42        # 1 batch of real (label=1.0) and fake images (label=0.0)
43        # randomly pick real images and
44        # corresponding labels from dataset
45        rand_indexes = np.random.randint(0,
46        train_size,
47                                size=batch_size)
48    real_images = x_train[rand_indexes]
49    real_labels = y_train[rand_indexes]
50    # generate fake images from noise using generator
51    # generate noise using uniform distribution
52    noise = np.random.uniform(-1.0,
53    1.0,
54                                size=[batch_size, latent_size])
55    # randomly pick one-hot labels
56    fake_labels = np.eye(num_labels)[np.random.choice(num_labels,
57    batch_size)]
58    # generate fake images
59    fake_images = generator.predict([noise, fake_labels]) # real + fake images = 1 batch of train data
60    x = np.concatenate((real_images, fake_images))
61    # real + fake labels = 1 batch of train data labels labels = np.concatenate((real_labels,

```



```

        fake_labels))
61 # label real and fake images
62 # real images label is 1.0
63 y = np.ones([2 * batch_size, 1])
64 # fake images label is 0.0
65 y[batch_size:, :] = 0
66 # train discriminator network, log the loss and accuracy # ['loss', 'activation_1_loss',
67 # 'label_loss', 'activation_1_acc', 'label_acc']
68 metrics = discriminator.train_on_batch(x, [y, labels]) fmt = "%d: [disc loss: %f, srcloss: %f,"
69 fmt += "lblloss: %f, srcacc: %f, lblacc: %f]"
70 log = fmt % (i, metrics[0], metrics[1], \
71             metrics[2], metrics[3], metrics[4])
72 # train the adversarial network for 1 batch
73 # 1 batch of fake images with label=1.0 and
74 # corresponding one-hot label or class
75 # since the discriminator weights are frozen
76 # in adversarial network only the generator is trained
77 # generate noise using uniform distribution
78 noise = np.random.uniform(-1.0,
79                             1.0,
80                             size=[batch_size, latent_size])
81
82 # randomly pick one-hot labels
83 fake_labels = np.eye(num_labels)[np.random.choice(num_labels,
84                                                    batch_size)]
85 # label fake images as real
86 y = np.ones([batch_size, 1])
87 # train the adversarial network
88 # note that unlike in discriminator training,
89 # we do not save the fake images in a variable
90 # the fake images go to the discriminator input
91 # of the adversarial for classification
92 # log the loss and accuracy
93 metrics = adversarial.train_on_batch([noise, fake_labels],
94                                     [y, fake_labels])
95
96 fmt = "%s [advr loss: %f, srcloss: %f,"
97 fmt += "lblloss: %f, srcacc: %f, lblacc: %f]"
98 log = fmt % (log, metrics[0], metrics[1], \
99             metrics[2], metrics[3], metrics[4])
100 print(log)
101 if (i + 1) % save_interval == 0:
102     # plot generator images on a periodic basis
103     gan.plot_images(generator,
104                     noise_input=noise_input,
105                     noise_label=noise_label,
106                     show=False,
107                     step=(i + 1),
108                     model_name=model_name)
109 # save the model after training the generator
110 # the trained generator can be reloaded
111 # for future MNIST digit generation
112 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 9.14: ACGAN 的输出样本

```

1 [0 1 2 3
2 4 5 6 7
3 8 9 0 1
4 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。

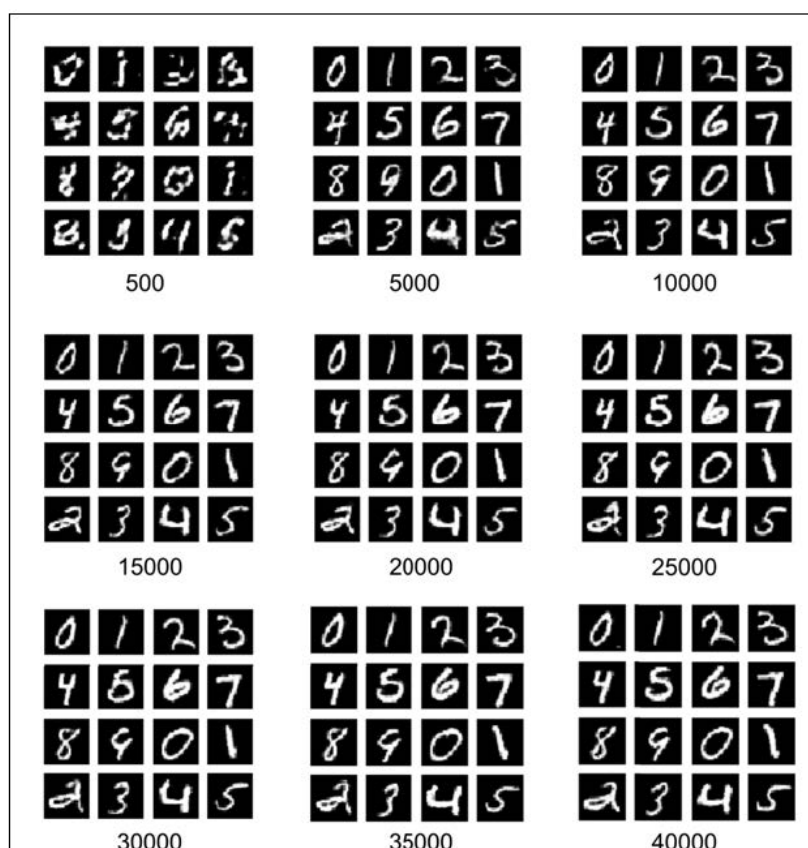


图 9.10: ACGAN 生成的样本输出是标签 [0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5] 的训练步骤的函数。

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。

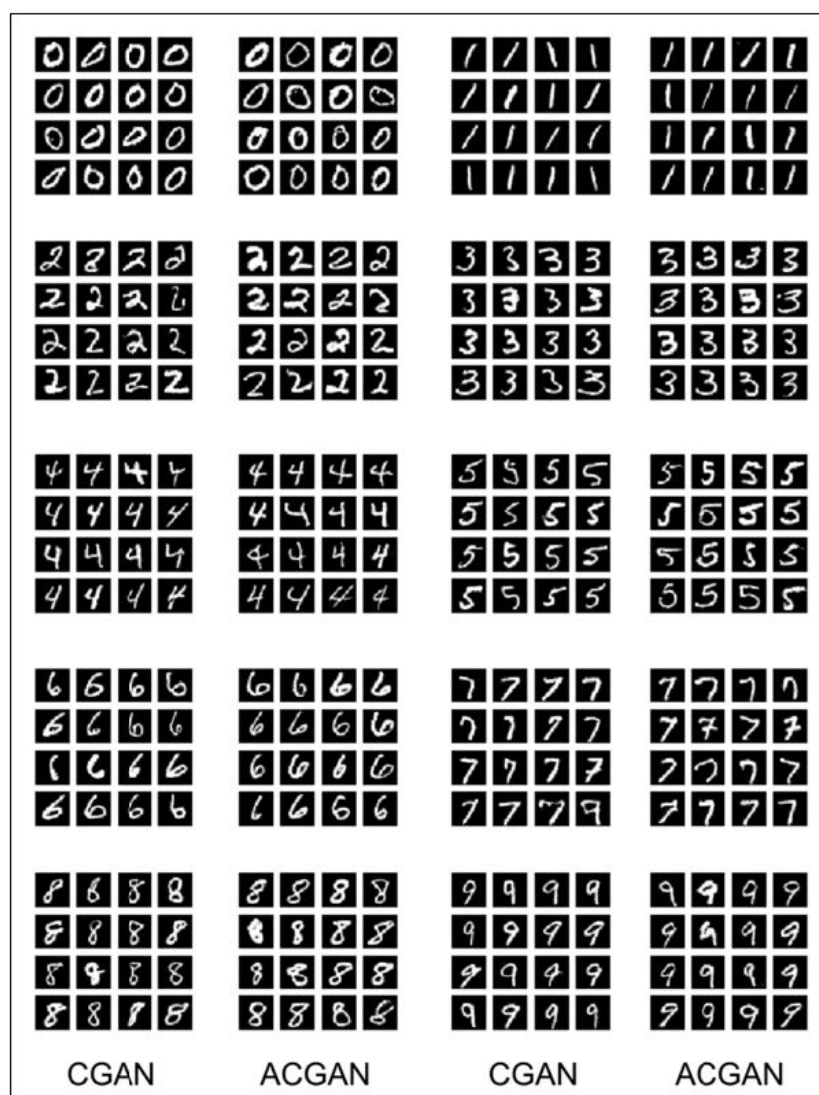


图 9.11: 对 CGAN 和 ACGAN 在数字 0 到 9 的条件下的输出进行并列比较

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

9.4. 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。

9.5. 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