11

跨域 GANs

在计算机视觉、计算机图形学和图像处理中，有许多任务涉及将图像从一种形式转化为另一种形式。灰度图像的着色、将卫星图像转换为地图、将一个艺术家的作品风格改变为另一个艺术家的作品风格、将夜间图像变成白天、将夏季照片变成冬季，这些只是几个例子。这些任务被称为跨域转移，将是本章的重点。源域中的图像被转移到目标域中，从而产生一个新的翻译图像。

跨域转移在现实世界中有许多实际应用。例如，在自动驾驶研究中，收集道路场景的驾驶数据既耗时又昂贵。在这个例子中，为了覆盖尽可能多的场景变化，道路会在不同的天气条件、季节和时间中被穿越，给我们带来大量不同的数据。通过使用跨域传输，我们有可能通过翻译现有的图像来生成新的合成场景，使其看起来真实。例如，我们可能只需要从一个地区收集夏季的道路场景，从另一个地方收集冬季的道路场景。然后，我们可以把夏天的图像转化为冬天的图像，把冬天的图像转化为夏天的图像。在这种情况下，需要完成的任务数量减少了一半。

生成逼真的合成图像是 GANs 擅长的一个领域。因此，跨域翻译是 GANs 的应用之一。在本章中，我们将重点讨论一种流行的跨域 GAN 算法，即 CycleGAN[2]。与其他跨域转换算法不同，例如 pix2pix[3]，CycleGAN 不需要对齐的训练图像来工作。在对齐的图像中，训练数据应该是由源图像和其相应的目标图像组成的一对图像；例如，一张卫星图像和由该图像衍生的相应地图。

CycleGAN 只需要卫星数据图像和地图。这些地图可能来自其他卫星数据，不一定是以前从训练数据中产生的。在本章中，我们将探讨以下内容：

- CycleGAN 的原理，包括它在 tf.keras 中的实现；
- CycleGAN 的应用实例，包括使用 CIFAR10 数据集对灰度图像进行着色，以及应用于 MNIST 数字和街景房号（SVHN）[1] 数据集的风格转移。

让我们先来谈谈 CycleGAN 背后的原理。

11.1. CycleGAN 的原理

将图像从一个领域翻译到另一个领域是计算机视觉、计算机图形学和图像处理中的一项常见任务。图11.1显示了边缘检测，这是一项常见的图像翻译任务。

图 11.1: 对齐图像的例子：左边是原始图像，右边是使用 Canny 边缘检测器转换的图像。原始照片是由作者拍摄的。

在这个例子中，我们可以把真实的照片（左）看作是源域的图像，而把边缘检测的照片（右）看作是目标域的样本。还有许多其他的跨域翻译程序有实际的应用，例如：

- 卫星图像到地图
- 脸部图像到表情符号、漫画或动漫
- 身体图像到头像
- 灰度照片的彩色化
- 医学扫描到真实照片
- 真实照片转为艺术家的画作

在不同领域还有很多这样的例子。例如，在计算机视觉和图像处理中，我们可以通过发明一种算法，从源图像中提取特征，将其翻译成目标图像，从而进行翻译。Canny 边缘算子就是这种算法的一个例子。然而，在许多情况下，翻译是非常复杂的手工工程，因此，几乎不可能找到一个合适的算法。源域和目标域的分布都是高维的和复杂的。

图像翻译问题的一个变通方法是使用深度学习技术。如果我们有足够大的源域和目标域的数据集，我们可以训练一个神经网络来为翻译建模。由于目标域的图像必须在给定的源图像中自动生成，它们必须看起来像目标域的真实样本。GANs 是一个适用于这种跨域任务的网络。pix2pix[3] 算法是一个跨域算法的例子。

pix2pix 算法与我们在第四章” 生成对抗网络（GANs）” 中讨论的条件性 GAN（CGAN）[4] 有相似之处。我们可以回顾一下，在 CGAN 中，在噪声输入 z 的基础上，有一个条件，如一个单热向量，制约着生成器的输出。例如，在 MNIST 数字中，如果我们想让生成器输出数字 8，条件是单热向量 [0, 0, 0, 0, 0, 0, 0, 1, 0] 。在 pix2pix 中，条件是要翻译的图像。生成器的输出是翻译后的图像。pix2pix 算法是通过优化 CGAN 损失来训练的。为了尽量减少生成的图像的模糊，L1 损失也被包括在内。

与 pix2pix 类似的神经网络的主要缺点是，训练输入和输出图像必须对齐。图 7.1.1 是一个对齐图像对的例子。目标图像的样本是由源图像生成的。在大多数情况下，对齐的图像对是不可用的，或者从源图像中生成的成本很高，或者我们不知道如何从给定的源图像中生成目标图像。我们所拥有的是源域和目标域的样本数据。图 7.1.2 是一个来自源域（真实照片）和目标域（梵高的艺术风格）的数据例子，是关于同一个向日葵的主题。源图像和目标图像不一定是对齐的。

与 pix2pix 不同，只要源数据和目标数据之间有足够的数量和变化，CycleGAN 就能学习图像翻译。不需要对齐。CycleGAN 学习源和目标分布，以及如何从给定的样本数据从源到目标分布进行翻译。不需要监督。

在图 7.1.2 的背景下，我们只需要数千张真实向日葵的照片和数千张梵高的向日葵画作的照片。训练完 CycleGAN 后，我们就能把向日葵的照片翻译成梵高的画。

图 11.2: 图片对不对齐的例子：左边是菲律宾大学大学路边的真实向日葵的照片，右边是英国伦敦国家美术馆文森特-梵高的《向日葵》。原始照片是由论文作者拍摄的。

下一个问题是：我们如何建立一个可以从非配对数据中学习的模型？在下一节中，我们将建立一个 CycleGAN，使用前向和后向循环 GAN 以及循环一致性检查来消除对成对输入数据的需求。

11.2. CycleGAN 模型

图11.3是 CycleGAN 的网络模型。

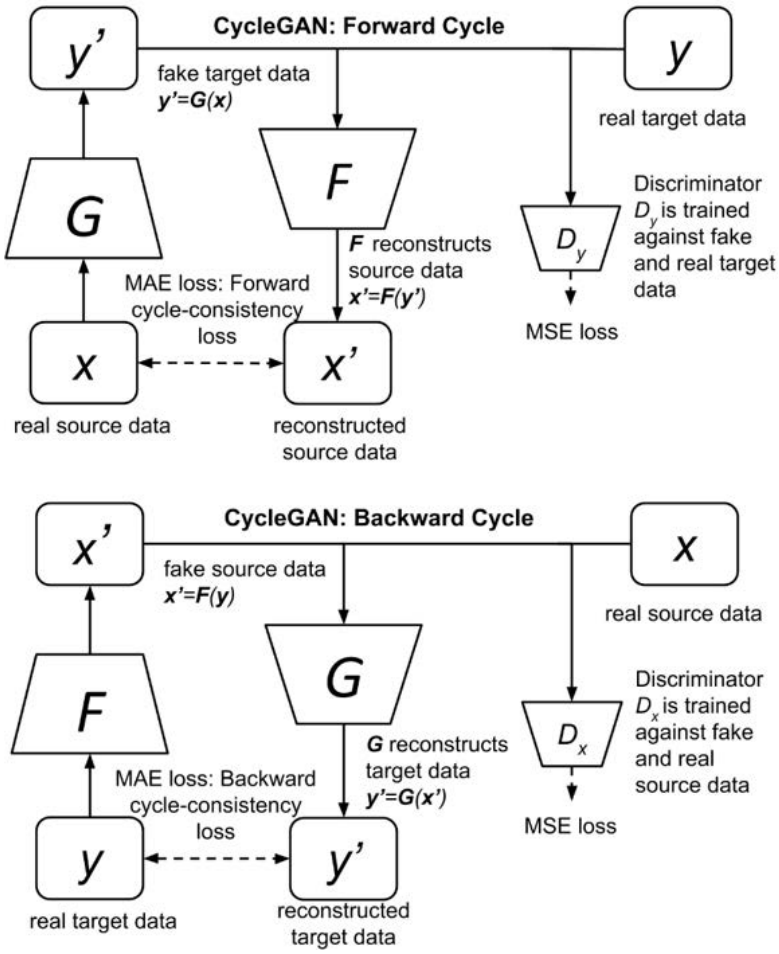

图 11.3: CycleGAN 模型包括四个网络。发生器 G、发生器 F、鉴别器 D_y 和鉴别器 D_x

让我们逐部分讨论图11.3。我们首先关注上层网络，也就是正向循环 GAN。如下图11.4所示，Forward CycleGAN 的目标是学习函数。

$$y' = G(x) \tag{11.1}$$

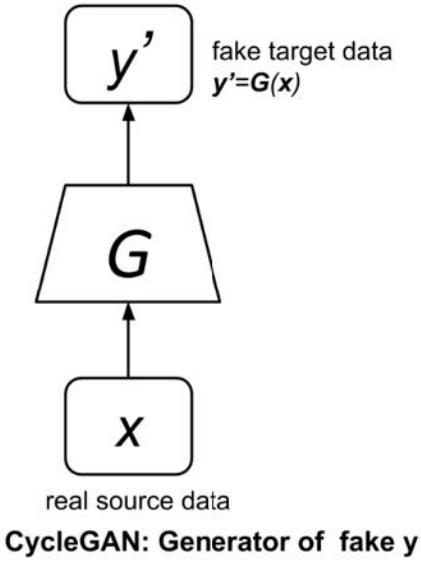

图 11.4: 假对象 Y 的 CycleGAN 发生器 G

公式11.1只是假目标数据的生成器 G。它将数据从源域 x 转换到目标域 y。为了训练生成器，我们必须建立一个 GAN。这就是图11.5中所示的前向循环 GAN。该图显示，它与第四章生成对抗网络（GAN）中的典型 GAN 一样，由生成器 G 和判别器 D_y 组成，可以以同样的对抗方式进行训练。学习是无监督的，只利用源域中现有的真实图像 x 和目标域中的真实图像 y。

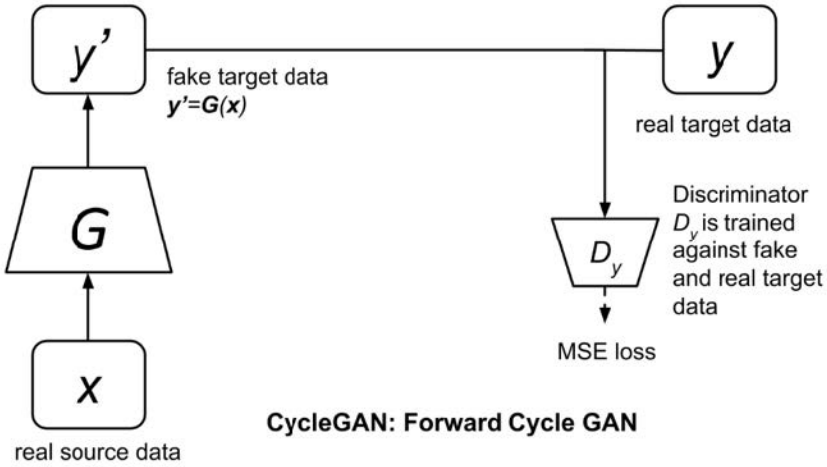

图 11.5: 前进的循环 GAN

与普通的 GANs 不同，CycleGAN 施加了周期一致性约束，如图11.6所示。前向周期一致性网络确保真实的源数据可以从假的目标数据中重构出来。

$$x' = F(G(x)) \tag{11.2}$$

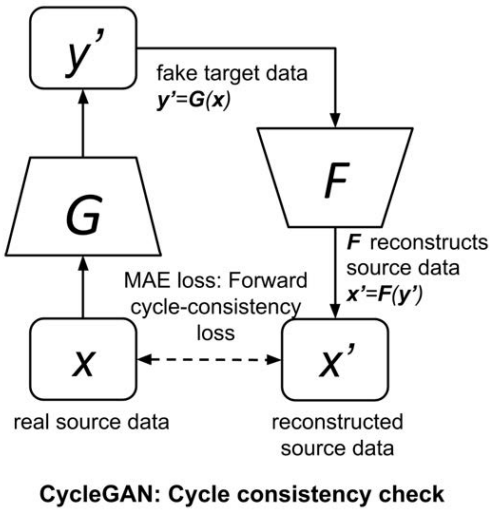

图 **11.6:** CycleGAN 的周期一致性检查

这是通过最小化前向周期一致性 $L1$ 损失来实现的，如下公式：

$$L_{forward-cyc} = \mathbb{E}_{x\ p_{data}(x)}||F(g(x)) - x||_1$$

循环一致性损失使用 $L1$，即平均绝对误差（MAE），因为与 $L2$，即平均平方误差（MSE）相比，它通常会导致较少的模糊图像重建。循环一致性检查意味着，尽管我们已经将源数据 x 转化为域 y，但 x 的原始特征应该在 y 中保持完整，并且可以恢复。网络 F 只是另一个生成器，我们将从后向循环 GAN 中借用，接下来将讨论。CycleGAN 是对称的。如图11.7所示，后向循环 GAN 与前向循环 GAN 相同，但源数据 x 和目标数据 y 的角色相反。源数据现在是 y，目标数据现在是 x。生成器 G 和 F 的角色也颠倒了。F 现在是生成器，而 G 恢复输入。在正向循环 GAN 中，生成器 F 是用于恢复源数据的网络，而 G 是生成器。

后向循环 GAN 发生器的目标是合成域，公式如下：

$$x' = F(y) \tag{11.4}$$

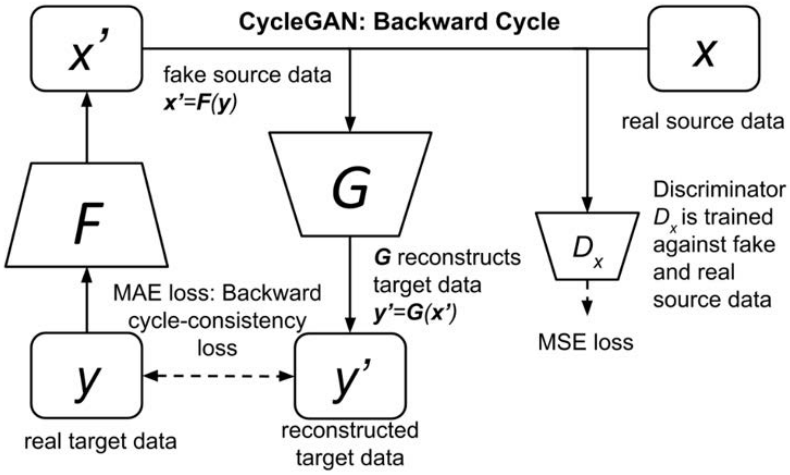

图 **11.7:** 向后循环 GAN

这是通过以对抗性方式训练后向循环 GAN 来实现的。其目的是让生成器 F 学会如何欺骗判别器，D_x。此外，还有一个类似的后向循环一致性的规定，以恢复原始源，y，数学表达如下：

$$y' = G(F(y)) \tag{11.5}$$

这是通过最小化后向周期一致性 $L1$ 损失来实现的。

$$L_{backward-cyc} = \mathbb{E}_{y\ p_{data}(y)} ||G(F(x)) - x||_1$$

总之，CycleGAN 的最终目标是让生成器 G 学会如何合成假的目标数据 y'，从而在前向循环中骗过判别器 D_y。由于网络是对称的，CycleGAN 还希望生成器 F 能够学会如何合成假的源数据 x'，以便在后向循环中骗过鉴别器 D_x。考虑到这一点，我们现在可以把所有的损失函数放在一起。

让我们从 GAN 的部分开始。受到最小二乘法 GAN（LSGAN）[5] 更好的感知质量的启发，如第五章改进的 GANs 中所述，CycleGAN 也使用 MSE 作为鉴别器和发生器的损失。回顾一下，LSGAN 和原始 GAN 的区别在于使用 MSE 损失而不是二进制交叉熵损失。

CycleGAN 将生成器-鉴别器的损失函数表示为。

$$L^{(D)}_{forward-GAN} = \mathbb{E}_{y\ p_{data}(y)}(D_y(y) - 1)^2 + \mathbb{E}_{x\ p_{data}(x)}(D_y(G(x))^2 \tag{11.7}$$

$$L^{(D)}_{forward-GAN} = \mathbb{E}_{x\ p_{data}(x)}(D_y(G(x)) - 1)^2 \tag{11.8}$$

$$L^{(D)}_{forward-GAN} = \mathbb{E}_{x\ p_{data}(x)}(D_x(x) - 1)^2 + \mathbb{E}_{y\ p_{data}(y)}(D_y(F(y))^2 \tag{11.9}$$

$$L^{(D)}_{forward-GAN} = \mathbb{E}_{y\ p_{data}(y)}(D_x(G(y)) - 1)^2 \tag{11.10}$$

$$L_{GAN}^{D} = L_{forward-GAN}^{D} + L_{backward-GAN}^{D} \tag{11.11}$$

$$L_{GAN}^{G} = L_{forward-GAN}^{G} + L_{backward-GAN}^{G} \tag{11.12}$$

第二组损失函数是周期一致性损失，可以通过将前向和后向 GANs 的贡献相加得出。

$$L_{cyc} = Lforward - cyc + L_{backward-cyc} \tag{11.13}$$

$$L_{cyc} = \mathbb{E}_{x\ p_{data}(x)}\left[\|F(G(x)) - x\|_1\right] + \mathbb{E}_{y\ p_{data}(y)}\left[\|F(G(y)) - y\|_1\right] \tag{11.14}$$

CycleGAN 的总损失为：

$$L = \lambda_1 L_{GAN} + \lambda_2 L_{cyc} \tag{11.15}$$

CycleGAN 推荐以下权重值，$\lambda_1 = 1.0$ 和 $\lambda_2 = 10.0$，以使循环一致性检查更加重要。
训练策略与 vanilla GAN 相似。Algorithm 7.1.1 概述了 CycleGAN 的训练过程。
Algorithm 7.1.1: CycleGAN 的训练过程:
重复 n 个训练步骤：

- 1. 通过使用真实的源数据和目标数据训练正向循环判别器来最小化 $L_{forward-GAN}^{(D)}$。一个真实目标数据 y，被标记为 1.0。一批假的目标数据 $y' = G(x)$，被标记为 0.0；
- 2. 通过使用真实的源数据和目标数据训练后向循环判别器，使 $L_{backward-GAN}^{(D)}$ 最小化。一个真实源数据 x，被标记为 1.0。一批假的源数据，$x' = F(y)$，被标记为 0.0；
- 3. 通过训练对抗网络中的"forward-cycle" 和"backward-cycle" 发生器，使 $L_{GAN}^{(G)}$ 和 L_{cyc} 最小化。一个假目标数据的 $y' = G(x)$，被标记为 1.0。一个假的源数据，$x' = F(y)$，被标记为 1.0。判别器的权重被冻结。

在 neural-style 转移问题中，色彩构成可能无法成功地从源图像转移到假目标图像。这个问题如图11.8所示。

图 11.8: 在风格转移过程中，颜色成分可能不会被成功转移。为了解决这个问题，身份损失被添加到总损失函数中

为了解决这个问题，CycleGAN 提出了包括前向和后向的循环认同损失函数，数学表达如下：

$$L_{identity} = \mathbb{E}_{x\ p_{data}(x)}\left[\|F(x) - x\|_1\right] + \mathbb{E}_{y\ p_{data}(y)}\left[\|G(y) - y\|_1\right] \tag{11.16}$$

总的 CycleGAN 损失的数学表达：

$$L = \lambda_1 L_{GAN} + \lambda_2 L_{cyc} + \lambda_3 L_{identity} \tag{11.17}$$

其中 $\lambda_3 = 0.5$ 。在对抗性训练中，身份损失也被优化。图11.9强调了 CycleGAN 实现身份正则器的辅助网络。

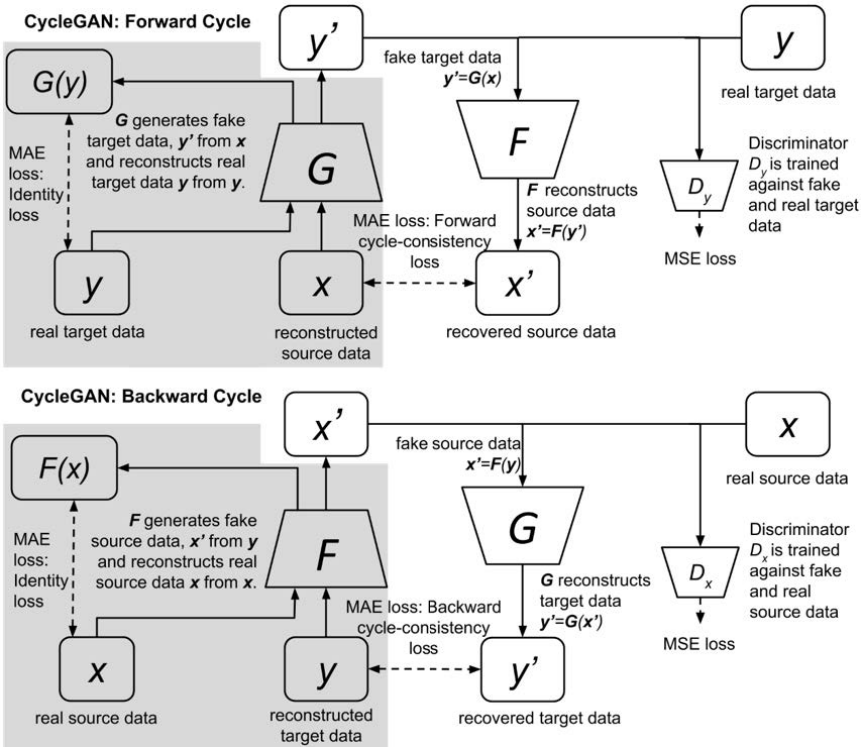

图 11.9: CycleGAN 模型的身份正则化网络在图像左侧突出显示。

在下一节中，我们将在 tf.keras 中实现 CycleGAN。

11.3. 使用 Keras 实现 CycleGAN

让我们来解决一个 CycleGAN 可以解决的简单问题。在第三章" 自动编码器" 中，我们用自动编码器对 CIFAR10 数据集中的灰度图像进行着色。我们可以回顾一下，CIFAR10 数据集包括 50,000 个训练过的数据项和 10,000 个测试数据样本，这些样本是属于 10 个类别的 32 x 32 RGB 图像。我们可以使用 rgb2gray（RGB）将所有彩色图像转换为灰度，如第三章自动编码器中所讨论的。

继而，我们可以将灰度训练图像作为源域图像，将原始彩色图像作为目标域图像。值得注意的是，虽然数据集是对齐的，但我们的 CycleGAN 的输入是彩色图像的随机样本和灰度图像的随机

样本。因此，我们的 CycleGAN 不会把训练数据看作是对齐的。训练结束后，我们将使用测试灰度图像来观察 CycleGAN 的性能。

正如前几节所讨论的，为了实现 CycleGAN，我们需要建立两个生成器和两个鉴别器。CycleGAN 的生成器学习源输入分布的潜在表示，并将这种表示翻译成目标输出分布。这正是自动编码器所做的。然而，典型的自动编码器与第三章" 自动编码器" 中讨论的自动编码器类似，使用一个编码器对输入进行降采样，直到瓶颈层，这时在解码器中的过程是相反的。

这种结构不适合某些图像转换问题，因为许多低层次的特征在编码器和解码器层之间是共享的。例如，在着色问题中，灰度图像的形式、结构和边缘都与彩色图像相同。为了规避这个问题，CycleGAN 生成器使用 U-Net[7] 结构，如图11.10所示。

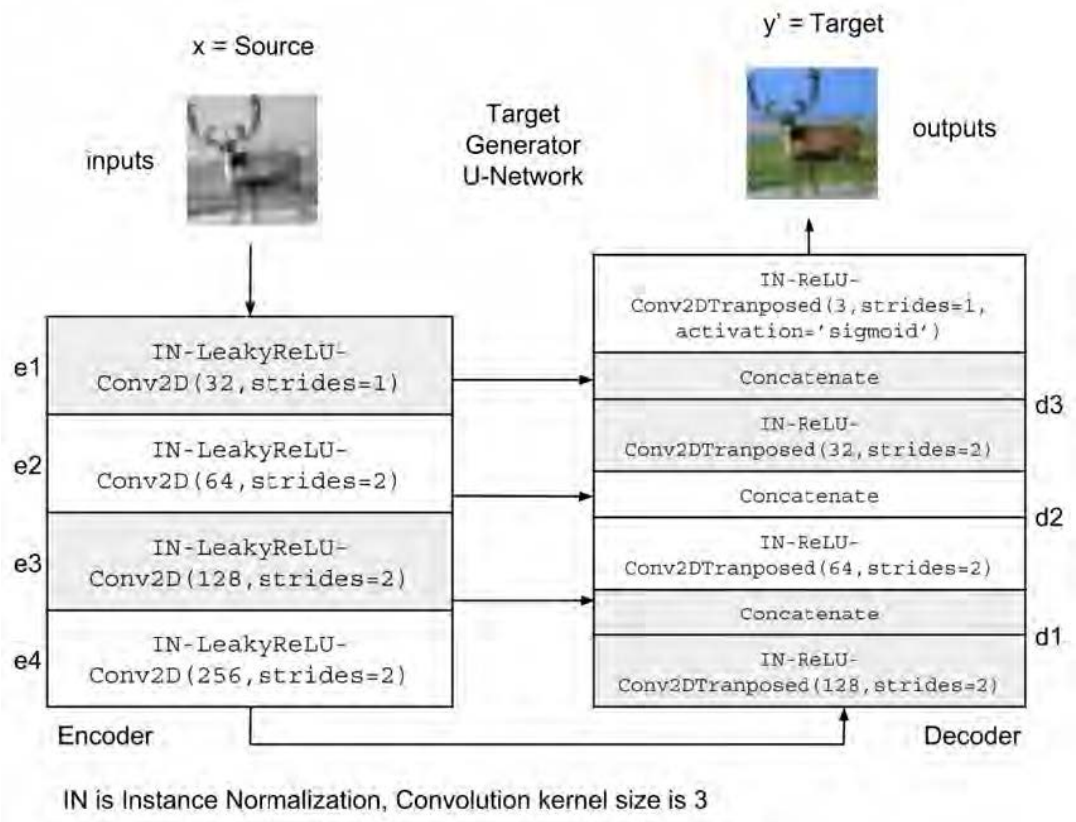

图 11.10: Keras 中前向循环发生器 G 的实现。该发生器是一个由编码器和解码器组成的 U 型网络 [7]。

在 U-Net 结构中，编码器层的输出 e_{n-i} 与解码器层的输出 d_i 相连接，其中 $n = 4$ 是编码器/解码器层的数量，$i = 1, 2, 3$ 是共享信息的层号。我们应该注意到，虽然这个例子使用了 $n = 4$，但输入/输出维度更高的问题可能需要更深的编码/解码层。U-Net 结构使特征级信息在编码器和解码器之间自由流动。编码器层由 Instance Normalization(IN)-LeakyReLU-Conv2D 组成，而解码器层则由 IN-ReLU-Conv2D 组成。编码器/解码器层的实现见清单 7.1.1，而发生器的实现见清单 7.1.2。[1]

实例归一化（IN）是对每个数据样本的批量归一化（BN）（也就是说，IN 是对每个图像或每个特征的 BN）。在样式转换中，重要的是对每个样本进行对比度归一化，而不是对每个批次进行归

[1]完整的代码可在 GitHub 上找到：https://github.com/PacktPublishing/Advanced-Deep-Learning-with-Keras

一化。IN 相当于对比度的归一化。同时，BN 打破了对比度的正常化。[2]

Listing 11.1: Listing 7.1.1: cyclegan-7.1.1.py

```
   def encoder_layer(inputs,
                     filters=16,
                     kernel_size=3,
                     strides=2,
                     activation='relu',
                     instance_norm=True):
       """Builds a generic encoder layer made of Conv2D-IN-LeakyReLU
       IN is optional, LeakyReLU may be replaced by ReLU
       """
       conv = Conv2D(filters=filters,
                  kernel_size=kernel_size,
                  strides=strides,
                  padding='same')
    x = inputs
    if instance_norm:
        x = InstanceNormalization(axis=3)(x)
    if activation == 'relu':
        x = Activation('relu')(x)
    else:
        x = LeakyReLU(alpha=0.2)(x)
    x = conv(x)
return x
def decoder_layer(inputs,
                  paired_inputs,
                  filters=16,
                  kernel_size=3,
                  strides=2,
                  activation='relu',
                  instance_norm=True):
    """Builds a generic decoder layer made of Conv2D-IN-LeakyReLU
    IN is optional, LeakyReLU may be replaced by ReLU
    Arguments: (partial)
    inputs (tensor): the decoder layer input
    paired_inputs (tensor): the encoder layer output
          provided by U-Net skip connection &
          concatenated to inputs.
"""
    conv = Conv2DTranspose(filters=filters,
                           kernel_size=kernel_size,
                           strides=strides,
                           padding='same')
    x = inputs
    if instance_norm:
        x = InstanceNormalization(axis=3)(x)
    if activation == 'relu':
        x = Activation('relu')(x)
    else:
        x = LeakyReLU(alpha=0.2)(x)
       x = conv(x)
       x = concatenate([x, paired_inputs])
       return x
```

[2]记得在使用 IN 之前安装 tensorflow-addons。$ pip install tensorflow-addons

接着是生成器的实现。Listing 7.1.2: cyclegan-7.1.1.py Keras 中的生成器实现

Listing 11.2: Listing 7.1.2: cyclegan-7.1.1.py

```
  def build_generator(input_shape,
                      output_shape=None,
                      kernel_size=3,
                      name=None):
      """The generator is a U-Network made of a 4-layer encoder
      and a 4-layer decoder. Layer n-i is connected to layer i.
      Arguments:
      input_shape (tuple): input shape
      output_shape (tuple): output shape
      kernel_size (int): kernel size of encoder & decoder layers
      name (string): name assigned to generator model
      Returns:
      generator (Model):
"""
      inputs = Input(shape=input_shape)
      channels = int(output_shape[-1])
      e1 = encoder_layer(inputs,
                         32,
                         kernel_size=kernel_size,
                         activation='leaky_relu',
                         strides=1)
      e2 = encoder_layer(e1,
                         64,
                         activation='leaky_relu',
                         kernel_size=kernel_size)
      e3 = encoder_layer(e2,
                         128,
                         activation='leaky_relu',
                         kernel_size=kernel_size)
      e4 = encoder_layer(e3,
                         256,
                         activation='leaky_relu',
                         kernel_size=kernel_size)
      d1 = decoder_layer(e4,
                         e3,
                         128,
                         kernel_size=kernel_size)
      d2 = decoder_layer(d1,
                         e2,
                         64,
                         kernel_size=kernel_size)
      d3 = decoder_layer(d2,
                         e1,
                         32,
                         kernel_size=kernel_size)
      outputs = Conv2DTranspose(channels,
                                kernel_size=kernel_size,
                                strides=1,
                                activation='sigmoid',
                                padding='same')(d3)
      generator = Model(inputs, outputs, name=name)
      return generator
```

CycleGAN 的判别器类似于普通的 GAN 判别器。输入的图像被多次降采样（在这个例子中是

三次）。最后一层是密集（1）层，它预测输入是真实的概率。除了不使用 IN 之外，每一层都与发生器的编码器层类似。然而，在大型图像中，用一个数字来计算图像是真的还是假的，结果是参数效率低下，导致生成器的图像质量差。

解决方案是使用 PatchGAN[6]，它将图像分为一个网格的斑块，并使用标量值的网格来预测斑块是真的概率。图11.11显示了 vanilla GAN 判别器和 2 x 2 PatchGAN 判别器的比较。

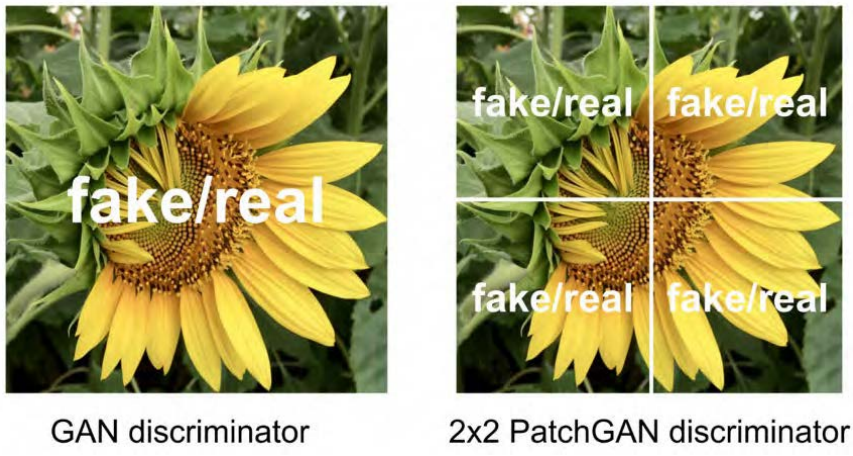

图 11.11: GAN 和 PatchGAN 判别器的比较

在这个例子中，这些斑块没有重叠，在它们的边界处相遇。然而，在一般情况下，斑块可能会重叠。

我们应该注意，PatchGAN 不是在 CycleGAN 中引入一种新的 GAN 类型。为了提高生成的图像质量，如果我们使用 2 x 2 的 PatchGAN，我们将有四个输出来进行判别，而不是一个输出。损失函数没有变化。直观地说，这是有道理的，因为如果图像的每个补丁或部分看起来都是真实的，那么整个图像看起来就会更真实。

图11.12显示了在 tf.keras 中实现的判别器网络。图中显示了判别器确定输入图像或一个补丁是彩色 CIFAR10 图像的可能性。

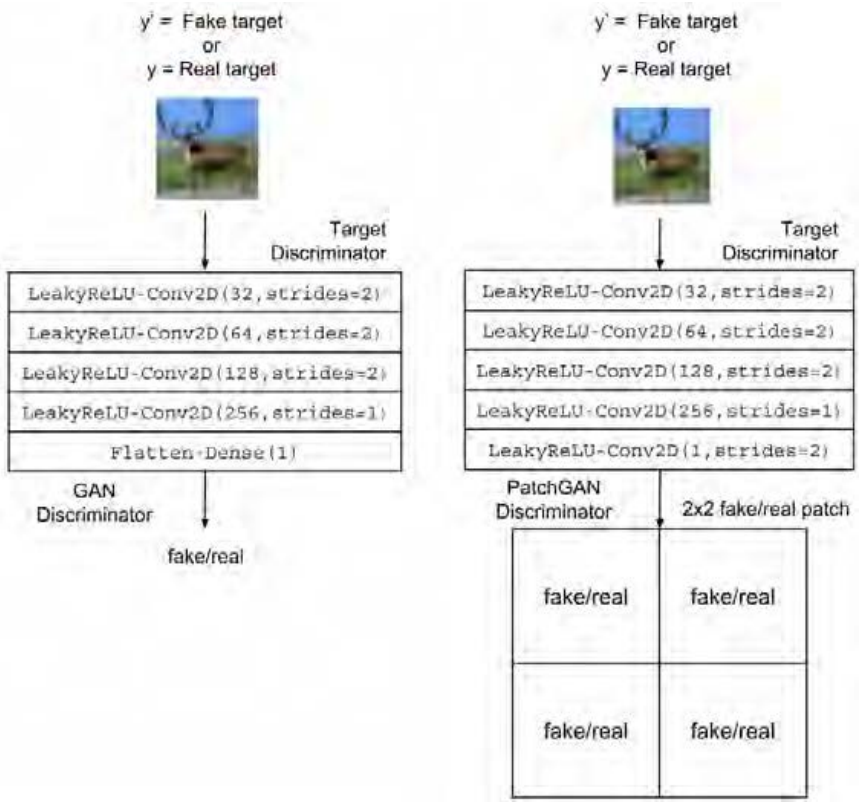

图 11.12: 目标判别器 D_y，在 tf.keras 中实现。PatchGAN 判别器显示在右边

由于输出的图像很小，只有 32 x 32 RGB，一个代表图像是真实的单一标量就足够了。然而，我们也评估了使用 PatchGAN 时的结果。Listing 7.1.3: cyclegan-7.1.1.py 鉴别器在 tf.keras 中的实现。

Listing 11.3: Listing 7.1.3: cyclegan-7.1.1.py

```
    def build_discriminator(input_shape,
                            kernel_size=3,
                            patchgan=True,
                            name=None):
"""The discriminator is a 4-layer encoder that outputs either
a 1-dim or a n x n-dim patch of probability that input is real
Arguments:
input_shape (tuple): input shape
kernel_size (int): kernel size of decoder layers
patchgan (bool): whether the output is a patch
    or just a 1-dim
name (string): name assigned to discriminator model
Returns:
discriminator (Model):
"""
inputs = Input(shape=input_shape)
x = encoder_layer(inputs,
                  32,
                  kernel_size=kernel_size,
                  activation='leaky_relu',
```

```
                  instance_norm=False)
x = encoder_layer(x,
                  64,
                  kernel_size=kernel_size,
                  activation='leaky_relu',
                  instance_norm=False)
x = encoder_layer(x,
                  128,
                  kernel_size=kernel_size,
                  activation='leaky_relu',
                  instance_norm=False)
x = encoder_layer(x,
                  256,
                  kernel_size=kernel_size,
                  strides=1,
                  activation='leaky_relu',
                  instance_norm=False)
# if patchgan=True use nxn-dim output of probability
# else use 1-dim output of probability
if patchgan:
           x = LeakyReLU(alpha=0.2)(x)
           outputs = Conv2D(1,
                            kernel_size=kernel_size,
                            strides=2,
                            padding='same')(x)
       else:
           x = Flatten()(x)
           x = Dense(1)(x)
           outputs = Activation('linear')(x)
       discriminator = Model(inputs, outputs, name=name)
       return discriminator
```

使用生成器和判别器构建器，我们现在能够构建 CycleGAN。清单 7.1.4 显示了构建器的功能。根据上一节的讨论，我们实例化了两个生成器，g_source = F 和 g_target = G，以及两个判别器，d_source = D_x 和 d_target = D_y。前向循环是：

$x' + F(G(x))$ =reco_source = g_source(g_target(source_input))。

向后的循环是：

$y' = G(F(y))$reco_target= g_target(g_source(target_input))。

对抗模型的输入是源和目标数据，而输出是 D_x 和 D_y 的输出以及重建的输入，x' 和 y'。由于灰度图像和彩色图像的通道数量不同，本例中没有使用身份网络。我们对 GAN 和循环一致性损失分别使用推荐的损失权重 $\lambda_1 = 1.0$ 和 $\lambda_2 = 10.0$。与前几章的 GAN 类似，我们使用 RMSprop，学习率为 $2e-4$，衰减率为 $6e-8$，用于判别器的优化器。对抗器的学习和衰减率是判别器的一半。

Listing 7.1.4: cyclegan-7.1.1.py tf.keras.CycleGAN 构建器:

Listing 11.4: Listing 7.1.4: cyclegan-7.1.1.py

```
  def build_cyclegan(shapes,
                     source_name='source',
                     target_name='target',
                     kernel_size=3,
                     patchgan=False,
                     identity=False
                     ):
      """Build the CycleGAN
```

```
        1) Build target and source discriminators
        2) Build target and source generators
        3) Build the adversarial network
        Arguments:
        shapes (tuple): source and target shapes
        source_name (string): string to be appended on dis/gen models
        target_name (string): string to be appended on dis/gen models
        kernel_size (int): kernel size for the encoder/decoder
            or dis/gen models
        patchgan (bool): whether to use patchgan on discriminator
        identity (bool): whether to use identity loss
        Returns:
        (list): 2 generator, 2 discriminator,
            and 1 adversarial models
        """
        source_shape, target_shape = shapes
        lr = 2e-4
        decay = 6e-8
        gt_name = "gen_" + target_name
        gs_name = "gen_" + source_name
        dt_name = "dis_" + target_name
        ds_name = "dis_" + source_name
        # build target and source generators
        g_target = build_generator(source_shape,
                                   target_shape,
                                   kernel_size=kernel_size,
                                   name=gt_name)
        g_source = build_generator(target_shape,
                                   source_shape,
                                   kernel_size=kernel_size,
                                   name=gs_name)
        print('---- TARGET GENERATOR ----')
        g_target.summary()
        print('---- SOURCE GENERATOR ----')
        g_source.summary()
        # build target and source discriminators
        d_target = build_discriminator(target_shape,
                                       patchgan=patchgan,
                                kernel_size=kernel_size,
                                name=dt_name)
d_source = build_discriminator(source_shape,
                                patchgan=patchgan,
                                kernel_size=kernel_size,
                                name=ds_name)
print('---- TARGET DISCRIMINATOR ----')
d_target.summary()
print('---- SOURCE DISCRIMINATOR ----')
d_source.summary()
optimizer = RMSprop(lr=lr, decay=decay)
d_target.compile(loss='mse',
                 optimizer=optimizer,
                 metrics=['accuracy'])
d_source.compile(loss='mse',
                 optimizer=optimizer,
                 metrics=['accuracy'])
d_target.trainable = False
d_source.trainable = False
```

```
# build the computational graph for the adversarial model
# forward cycle network and target discriminator
source_input = Input(shape=source_shape)
fake_target = g_target(source_input)
preal_target = d_target(fake_target)
reco_source = g_source(fake_target)
# backward cycle network and source discriminator
target_input = Input(shape=target_shape)
fake_source = g_source(target_input)
preal_source = d_source(fake_source)
reco_target = g_target(fake_source)
# if we use identity loss, add 2 extra loss terms
# and outputs
if identity:
    iden_source = g_source(source_input)
    iden_target = g_target(target_input)
    loss = ['mse', 'mse', 'mae', 'mae', 'mae', 'mae']
    loss_weights = [1., 1., 10., 10., 0.5, 0.5]
    inputs = [source_input, target_input]
    outputs = [preal_source,
                      preal_target,
                      reco_source,
                      reco_target,
                      iden_source,
                      iden_target]
       else:
           loss = ['mse', 'mse', 'mae', 'mae']
           loss_weights = [1., 1., 10., 10.]
           inputs = [source_input, target_input]
           outputs = [preal_source,
                      preal_target,
                      reco_source,
                      reco_target]
       # build adversarial model
       adv = Model(inputs, outputs, name='adversarial')
       optimizer = RMSprop(lr=lr*0.5, decay=decay*0.5)
       adv.compile(loss=loss,
                   loss_weights=loss_weights,
                   optimizer=optimizer,
                   metrics=['accuracy'])
       print('---- ADVERSARIAL NETWORK ----')
       adv.summary()
       return g_source, g_target, d_source, d_target, adv
```

我们遵循上一节算法 7.1.1 中的训练程序，我们可以回忆一下。清单 7.1.5 显示了 CycleGAN 的训练。这个训练与普通 GAN 的细微差别在于有两个判别器需要优化。然而，只有一个对抗性模型需要优化。每 2,000 步，生成器就会保存预测的源图像和目标图像。我们将使用 32 的批次大小。我们还尝试了 1 的批处理量，但输出质量几乎相同，而且需要更长的时间来训练（在 NVIDIA GTX 1060 上，批处理量为 1 的 43 毫秒/图像，而批处理量为 32 的 3.6 毫秒/图像）。

tf.keras 中的 CycleGAN 训练程序:

Listing 11.5: Evaluate how the model does on the test set

```
  def train_cyclegan(models,
                     data,
                     params,
```

```
                      test_params,
                      test_generator):
        """ Trains the CycleGAN.
1) Train the target discriminator
2) Train the source discriminator
3) Train the forward and backward cyles of
    adversarial networks
Arguments:
models (Models): Source/Target Discriminator/Generator,
    Adversarial Model
data (tuple): source and target training data
params (tuple): network parameters
test_params (tuple): test parameters
test_generator (function): used for generating
    predicted target and source images
"""
# the models
g_source, g_target, d_source, d_target, adv = models
# network parameters
batch_size, train_steps, patch, model_name = params
# train dataset
source_data, target_data, test_source_data, test_target_data\
        = data
titles, dirs = test_params
# the generator image is saved every 2000 steps
save_interval = 2000
target_size = target_data.shape[0]
source_size = source_data.shape[0]
# whether to use patchgan or not
if patch > 1:
    d_patch = (patch, patch, 1)
    valid = np.ones((batch_size,) + d_patch)
    fake = np.zeros((batch_size,) + d_patch)
else:
    valid = np.ones([batch_size, 1])
    fake = np.zeros([batch_size, 1])
valid_fake = np.concatenate((valid, fake))
start_time = datetime.datetime.now()
for step in range(train_steps):
            # sample a batch of real target data
            rand_indexes = np.random.randint(0,
                                             target_size,
                                             size=batch_size)
            real_target = target_data[rand_indexes]
            # sample a batch of real source data
            rand_indexes = np.random.randint(0,
                                             source_size,
                                             size=batch_size)
            real_source = source_data[rand_indexes]
            # generate a batch of fake target data fr real source data
            fake_target = g_target.predict(real_source)
            # combine real and fake into one batch
            x = np.concatenate((real_target, fake_target))
            # train the target discriminator using fake/real data
            metrics = d_target.train_on_batch(x, valid_fake)
            log = "%d: [d_target loss: %f]" % (step, metrics[0])
            # generate a batch of fake source data fr real target data
```

```
            fake_source = g_source.predict(real_target)
            x = np.concatenate((real_source, fake_source))
            # train the source discriminator using fake/real data
            metrics = d_source.train_on_batch(x, valid_fake)
            log = "%s [d_source loss: %f]" % (log, metrics[0])
            # train the adversarial network using forward and backward
            # cycles. the generated fake source and target
            # data attempts to trick the discriminators
            x = [real_source, real_target]
            y = [valid, valid, real_source, real_target]
            metrics = adv.train_on_batch(x, y)
            elapsed_time = datetime.datetime.now() - start_time
            fmt = "%s [adv loss: %f] [time: %s]"
            log = fmt % (log, metrics[0], elapsed_time)
            print(log)
            if (step + 1) % save_interval == 0:
                test_generator((g_source, g_target),
                               (test_source_data, test_target_data),
                               step=step+1,
                               titles=titles,
                               dirs=dirs,
                               show=False)
        # save the models after training the generators
        g_source.save(model_name + "-g_source.h5")
        g_target.save(model_name + "-g_target.h5")
```

Finally, before we can use the CycleGAN to build and train functions, we have to perform some data preparation. The modules cifar10_utils.py and other_ utils.py load the CIFAR10 training and test data. Please refer to the source code for details of these two files. After loading, the train and test images are converted to grayscale to generate the source data and test source data.

Listing 7.1.6 shows how the CycleGAN is used to build and train a generator network (g_target) for colorization of grayscale images. Since CycleGAN is symmetric, we also build and train a second generator network (g_source) that converts from color to grayscale. Two CycleGAN colorization networks were trained. The first uses discriminators with a scalar output similar to vanilla GAN, while the second uses a 2 x 2 PatchGAN.

Listing 7.1.6: cyclegan-7.1.1.py CycleGAN for colorization:

```
   def graycifar10_cross_colorcifar10(g_models=None):
        """Build and train a CycleGAN that can do
            grayscale <--> color cifar10 images
        """
        model_name = 'cyclegan_cifar10'
        batch_size = 32
        train_steps = 100000
        patchgan = True
        kernel_size = 3
        postfix = ('%dp' % kernel_size) \
                if patchgan else ('%d' % kernel_size)
        data, shapes = cifar10_utils.load_data()
        source_data, _, test_source_data, test_target_data = data
        titles = ('CIFAR10 predicted source images.',
                  'CIFAR10 predicted target images.',
                  'CIFAR10 reconstructed source images.',
                  'CIFAR10 reconstructed target images.')
        dirs = ('cifar10_source-%s' % postfix, \
                'cifar10_target-%s' % postfix)
        # generate predicted target(color) and source(gray) images
        if g_models is not None:
```

```
            g_source, g_target = g_models
            other_utils.test_generator((g_source, g_target),
return
        # build the cyclegan for cifar10 colorization
        models = build_cyclegan(shapes,
                                "gray-%s" % postfix,
                                "color-%s" % postfix,
                                kernel_size=kernel_size,
                                patchgan=patchgan)
        # patch size is divided by 2^n since we downscaled the input
        # in the discriminator by 2^n (ie. we use strides=2 n times)
        patch = int(source_data.shape[1] / 2**4) if patchgan else 1
        params = (batch_size, train_steps, patch, model_name)
        test_params = (titles, dirs)
        # train the cyclegan
        train_cyclegan(models,
                       data,
                       params,
                       test_params,
                       other_utils.test_generator)
```

在下一节中，我们将研究 CycleGAN 的生成器输出，用于着色。

11.4. Generator outputs of CycleGAN

图11.13显示了 CycleGAN 的着色结果。源图像来自测试数据集。

图 11.13: 使用不同的技术进行着色效果

为了进行比较，我们展示了使用第三章" 自动编码器" 中描述的普通自动编码器的基础真相和着色结果。一般来说，所有着色的图像在感知上都是可以接受的。总的来说，似乎每种着色技术都有自己的优点和缺点。所有的着色方法对天空和车辆的正确颜色都不一致。

例如，飞机背景中的天空（第三行，第二列）是白色的。自动编码器得到了正确的结果，但 CycleGAN 认为它是浅棕色或蓝色。

对于第六行第六列，黑暗的海面上的船有一个阴暗的天空，但被自动编码器着色为蓝色的天空和蓝色的海，而 CycleGAN 在没有 PatchGAN 的情况下将蓝色的海和白色的天空着色。这两种预测在现实世界中都是有意义的。同时，带有 PatchGAN 的 CycleGAN 的预测与地面真相相似。在倒数第二行和第二列，没有任何方法能够预测汽车的红色。在动物身上，CycleGAN 的两种口味的颜色都接近于地面真相。

由于 CycleGAN 是对称的，它也能预测给定的彩色图像的灰度图像。图 7.1.14 显示了两种 CycleGAN 变体进行的颜色到灰度的转换。目标图像来自测试数据集。除了一些图像的灰度色调有细微差别外，预测一般都很准确。

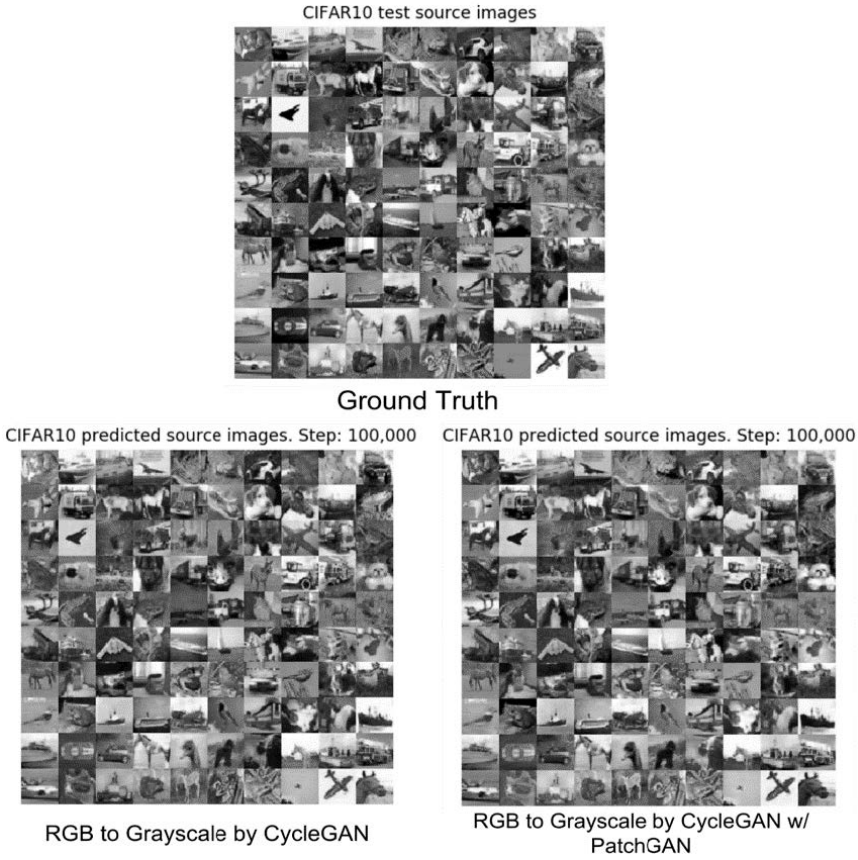

图 11.14: 彩色 (图11.9) 到灰度的转换

为了训练 CycleGAN 的着色，命令是：python3 cyclegan-7.1.1.py -c 读者可以通过使用 CycleGAN 与 PatchGAN 的预训练模型来运行图像翻译。

python3 cyclegan-7.1.1.py –cifar10_g_source=cyclegan_cifar10-g_source.h5–cifar10_g_target=cyclegan_cifar10-g_target.h5

在本节中，我们展示了 CycleGAN 在着色方面的一个实际应用。在下一节中，我们将在更具挑战性的数据集上训练 CycleGAN。源域 MNIST 与目标域 SVHN 数据集 [1] 有很大的不同。

11.5. MNIST 和 SVHN 数据集上的 CycleGAN

我们现在要解决的是一个更具挑战性的问题。假设我们使用灰度的 MNIST 数字作为我们的源数据，而我们想从 SVHN[1] 中借用风格，这是我们的目标数据。每个领域的样本数据如图11.15所示。

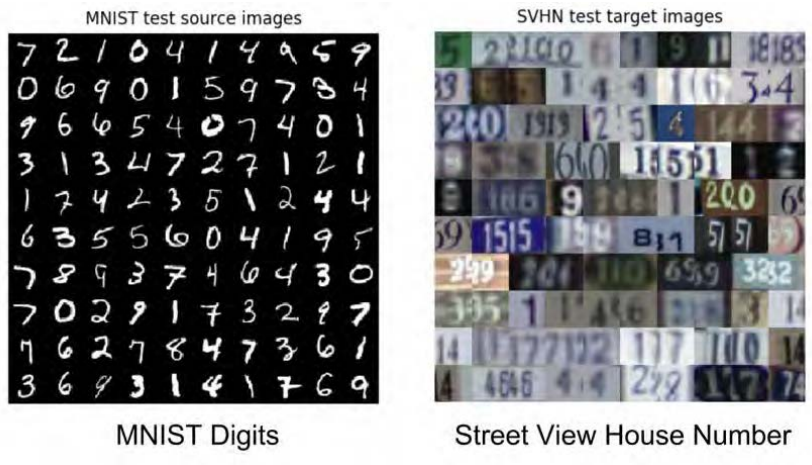

图 11.15: 两个不同的数据集

我们可以重复使用上一节中讨论过的 CycleGAN 的所有构建和训练函数来执行风格转换。唯一的区别是，我们必须增加加载 MNIST 和 SVHN 数据的程序。[3] 我们引入 mnist_svhn_utils.py 模块来帮助我们完成这项任务。Listing7.1.7 显示了跨域转移的 CycleGAN 的初始化和训练。CycleGAN 的结构与上一节相同，只是我们使用了 5 个内核的大小，因为两个域有很大的不同。

Listing 7.1.7: cyclegan-7.1.1.py CycleGAN 用于 MNIST 和 SVHN 之间的跨域风格转换。

Listing 11.6: Listing 7.1.7: cyclegan-7.1.1.py

```
    def mnist_cross_svhn(g_models=None):
        """Build and train a CycleGAN that can do mnist <--> svhn
        """
        model_name = 'cyclegan_mnist_svhn'
        batch_size = 32
        train_steps = 100000
        patchgan = True
        kernel_size = 5
        postfix = ('%dp' % kernel_size) \
                if patchgan else ('%d' % kernel_size)
        data, shapes = mnist_svhn_utils.load_data()
        source_data, _, test_source_data, test_target_data = data
        titles = ('MNIST predicted source images.',
                  'SVHN predicted target images.',
                  'MNIST reconstructed source images.',
                  'SVHN reconstructed target images.')
        dirs = ('mnist_source-%s' \
                % postfix, 'svhn_target-%s' % postfix)
# generate predicted target(svhn) and source(mnist) images
if g_models is not None:
    g_source, g_target = g_models
    other_utils.test_generator((g_source, g_target),
return
        # build the cyclegan for mnist cross svhn
        models = build_cyclegan(shapes,
                                "mnist-%s" % postfix,
                                "svhn-%s" % postfix,
```

[3] SVHN 的数据集可以在 http://ufldl.stanford.edu/housenumbers/。

```
                                kernel_size=kernel_size,
                                patchgan=patchgan)
        # patch size is divided by 2^n since we downscaled the input
        # in the discriminator by 2^n (ie. we use strides=2 n times)
        patch = int(source_data.shape[1] / 2**4) if patchgan else 1
        params = (batch_size, train_steps, patch, model_name)
        test_params = (titles, dirs)
        # train the cyclegan
        train_cyclegan(models,
                       data,
                       params,
                       test_params,
                       other_utils.test_generator)
```

将测试数据集中的 MNIST 转移到 SVHN 的结果如图11.16所示。生成的图像具有 SVHN 的风格，但是数字并没有完全转移。例如，在第四行，数字 3、1 和 3 被 CycleGAN 风格化。然而，在第三行，数字 9、6 和 6 在没有 CycleGAN 和有 PatchGAN 的情况下分别被风格化为 0、6、01、0、65 和 68。

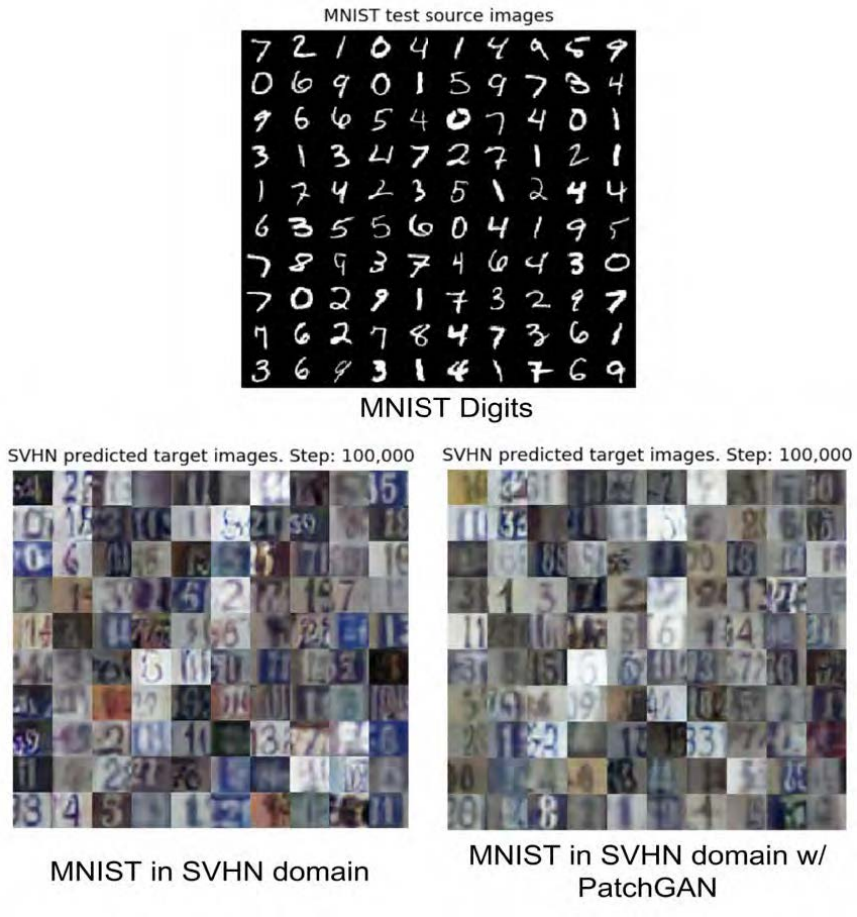

图 11.16: 将测试数据从 MNIST 域转移到 SVHN 的样式

后向循环的结果如图 7.1.17 所示。在这种情况下，目标图像是来自 SVHN 测试数据集。生成

的图像具有 MNIST 的风格，但是数字的翻译并不正确。例如，在第一行，数字 5、2 和 210 在没有 CycleGAN 和有 PatchGAN 的情况下分别被风格化为 7、7、8、3、3 和 1。

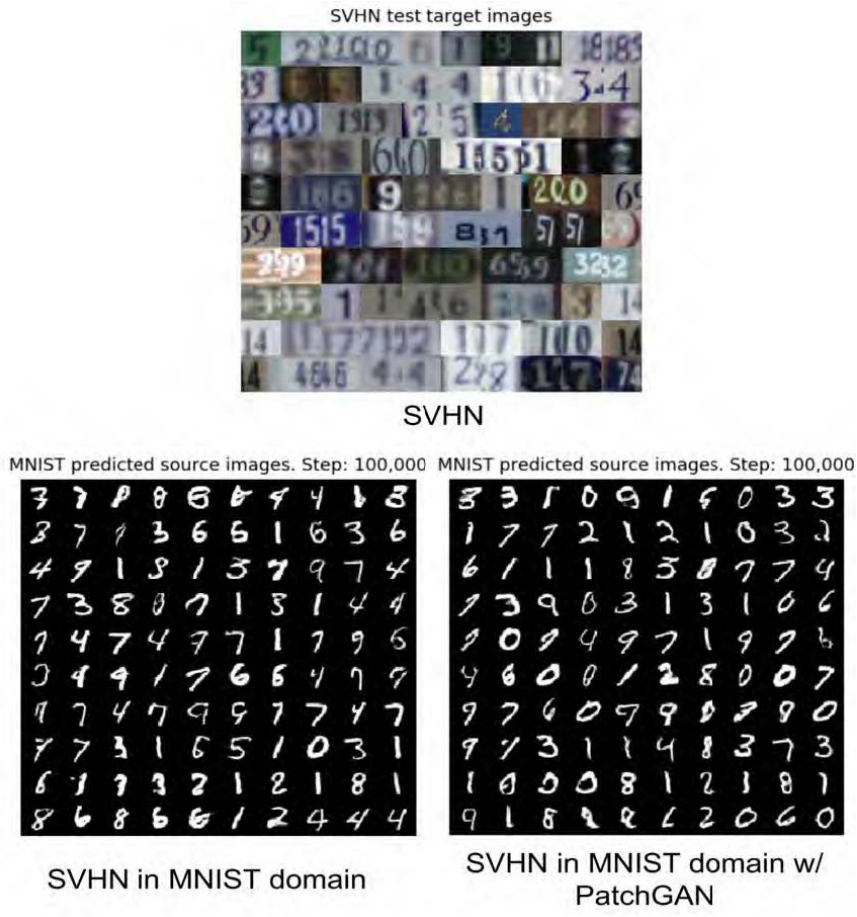

图 11.17: 将测试数据从 SVHN 域转移到 MNIST 的样式

在 PatchGAN 的情况下，鉴于预测的 MNIST 数字被限制为一个数字，输出 1 是可以理解的。有一些正确的预测，比如在第二行，SVHN 数字的最后三列，6、3、4 被 CycleGAN 转换为 6、3、6，而没有 PatchGAN。然而，CycleGAN 的两种类型的输出都是一致的个位数，并且可以识别。在从 MNIST 到 SVHN 的转换中表现出来的问题，即源域的一个数字被翻译成目标域的另一个数字，被称为标签翻转 [8]。尽管 CycleGAN 的预测是周期一致的，但它们不一定是语义一致的。数字的含义在翻译过程中会丢失。

为了解决这个问题，霍夫曼 [8] 引入了一个改进的 CycleGAN，称为循环一致的对抗域适应（CyCADA）。不同的是，额外的语义损失项确保预测不仅是周期一致的，而且是语义一致的。图11.18显示了 CycleGAN 在正向循环中重建 MNIST 数字的情况。重建的 MNIST 数字与源 MNIST 数字几乎完全相同。

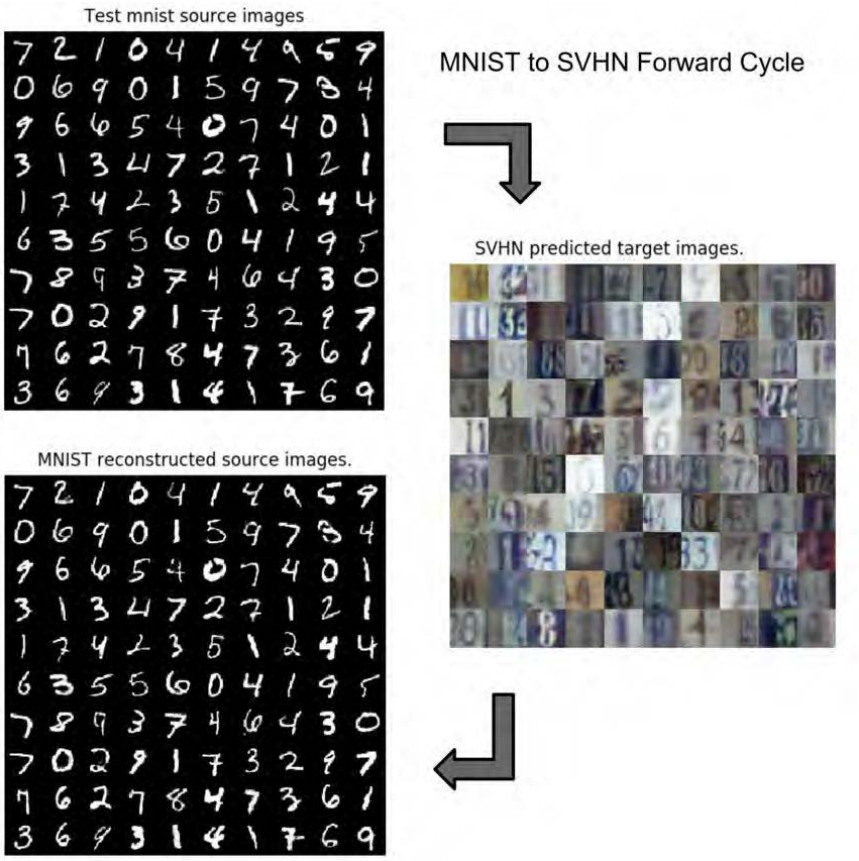

图 **11.18:** CycleGAN 与 PatchGAN 在 MNIST（源）到 SVHN（目标）的正向循环。

图11.19显示了 CycleGAN 在后向循环中对 SVHN 数字的重建。

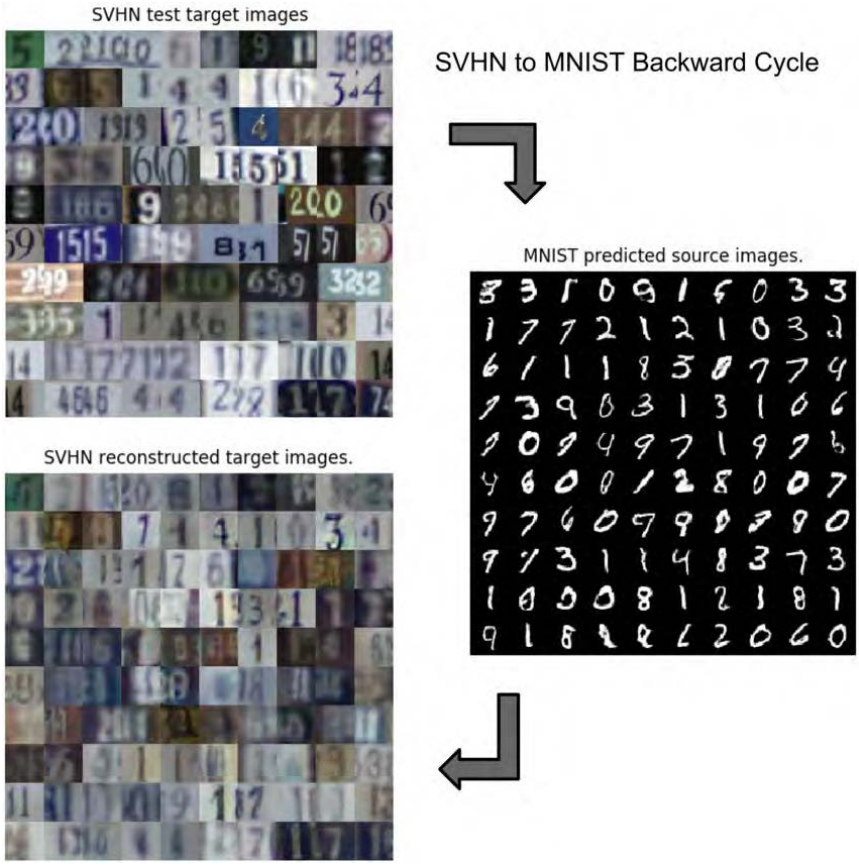

图 **11.19:** CycleGAN 与 PatchGAN 在 MNIST（源）到 SVHN（目标）的后向循环。

在图11.3中，CycleGAN 被描述为周期一致。换句话说，给定源 x，CycleGAN 在前向循环中重构源为 x'。此外，给定目标 y，CycleGAN 在后向循环中重建目标为 y'。

许多目标图像被重建了。有些数字显然是相同的，如第二行，在最后两列（3 和 4），而有些数字是相同的，但模糊不清，如第一行，在前两列（5 和 2）。有些数字被转换为另一个数字，尽管风格仍然像第二行，在前两列（从 33 和 6 到 1 和一个无法识别的数字）。

为了将 MNIST 的 CycleGAN 训练成 SVHN，命令是：

```
python3 cyclegan-7.1.1.py -m
```

我们鼓励读者使用 CycleGAN 与 PatchGAN 的预训练模型来运行图像翻译。

```
python3 cyclegan-7.1.1.py –mnist_svhn_g_source=cyclegan_mni-
st_svhn-g_source.h5 –mnist_svhn_g_target=cyclegan_mnist_svhn-g_target.h5
```

到目前为止，我们只看到 CycleGAN 的两个实际应用。两者都是在小数据集上演示的，以强调可重复性的概念。

正如本章前面提到的，CycleGAN 还有许多其他的实际应用。我们在这里介绍的 CycleGAN 可以作为具有更大分辨率的图像翻译的基础。

11.6. 结束语

在这一章中，我们已经讨论了 CycleGAN 作为一种可以用于图像翻译的算法。在 CycleGAN 中，源数据和目标数据不一定是对齐的。我们演示了两个例子，灰度 ↔ 彩色，以及 MNIST ↔ SVHN，尽管 CycleGAN 还可以进行许多其他可能的图像转换。

11.7. 参考文献

1. Yuval Netzer et al.: Reading Digits in Natural Images with Unsupervised Feature Learning. NIPS workshop on deep learning and unsupervised feature learning. Vol. 2011. No. 2. 2011

2. Zhu-Jun-Yan et al.: Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. 2017 IEEE International Conference on Computer Vision (ICCV). IEEE, 2017

3. Phillip Isola et al.: Image-to-Image Translation with Conditional Adversarial Networks. 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). IEEE, 2017

4. Mehdi Mirza and Simon Osindero. Conditional Generative Adversarial Nets.

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

6. Chuan Li and Michael Wand. Precomputed Real-Time Texture Synthesis with Markovian Generative Adversarial Networks. European Conference on Computer Vision. Springer, Cham, 2016

7. Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. International Conference on Medical image computing and computer-assisted intervention. Springer, Cham, 2015

8. Judy Hoffman et al.: CyCADA: Cycle-Consistent Adversarial Domain Adaptation. arXiv preprint arXiv:1711.03213, 2017