6

自动编码器

自动编码器，它是一种神经网络架构，试图找到给定输入数据的压缩表示。

输入数据可能是多种形式的，包括语音、文本、图像或视频。一个自动编码器将试图找到一个表示或一段代码，以便对输入数据进行有用的转换。举例来说，当去噪自动编码器时，神经网络将试图找到一个可以用来将噪声数据转化为干净数据的代码。噪声数据的形式可以是带有静态噪声的音频记录，然后将其转换为清晰的声音。自动编码器将仅从数据中自动学习代码，而不需要人工标注。因此，自动编码器可以被归入无监督学习算法。

在本书后面的章节中，我们将研究生成对抗网络（GANs）和变异自动编码器（VAEs），它们也是无监督学习算法的代表形式。这与我们在前几章中讨论的监督学习算法形成鲜明对比，后者需要人类的注释。

综上所述，本章介绍了：- 自动编码器的原理 - 如何使用 tf.keras 实现自动编码器 - 去噪和着色自动编码器的实际应用

首先来了解一下什么是自动编码器，以及自动编码器的原理。

6.1. 自动编码器的原理

在其最简单的形式中，自动编码器将通过尝试将输入复制到输出来学习表示或代码。然而，使用自动编码器并不像把输入复制到输出那么简单。否则，神经网络将无法发现输入分布中的隐藏结构。

自动编码器将把输入分布编码成一个低维的张量，它通常采用矢量的形式。这将近似于隐藏的结构，通常被称为潜在的表示、代码或矢量。这个过程构成了编码部分。然后，潜伏向量将被解码器部分解码以恢复原始输入。

由于潜伏向量是输入分布的低维压缩表示，应该可以预见，解码器恢复的输出只能与输入相近。输入和输出之间的不相似性可以用损失函数来衡量。

但我们为什么要使用自动编码器呢？简单地说，自动编码器在其原始形式或作为更复杂的神经网络的一部分都有实际应用。

它们是理解深度学习高级课题的关键工具，因为它们给了我们一个适合密度估计的低维数据

表示。此外，它可以被有效地处理，以对输入数据进行结构性操作。常见的操作包括去噪、着色、特征级运算、检测、跟踪和分割，这只是其中的几个例子。

在本节中，我们将介绍自动编码器的原理。我们将用前几章介绍的 MNIST 数据集来研究自动编码器。

首先，我们需要知道，自动编码器有两个运算符，它们是：

- 编码器（Encoder）：它将输入 x 转化为一个低维的潜向量 $z = f(x)$。由于潜伏向量是低维的，编码器被迫只学习输入数据中最重要的特征。例如，在 MNIST 数字的情况下，要学习的重要特征可能包括书写方式、倾斜角度、笔画的圆度、厚度等等。从本质上讲，这些都是表示零到九的数字所需的最重要的信息位。

- 解码器（Decoder）：它试图从潜伏矢量中恢复输入，$g(z) = x$。

虽然潜伏向量的维度很低，但它有足够的大小，使解码器能够恢复输入数据。

解码器的目标是使 x 尽可能地接近 x。一般来说，编码器和解码器都是非线性函数。z 的维度是衡量它所能代表的突出特征的数量。为了提高效率，也为了约束潜伏代码只学习输入分布中最突出的属性，该维度通常比输入维度小得多 [1]。

当潜伏代码的维度明显大于 x 时，自动编码器有记忆输入的倾向。

一个合适的损失函数，$L(x, x)$，是衡量输入 x 与输出，即恢复的输入，x 的相似程度。如下式所示，平均平方误差（MSE）是这种损失函数的一个例子。

$$L(x, x) = MSE = \frac{m}{1} \sum_{i=1} i = m(x_1 - x_2)^2 \tag{6.1}$$

在这个例子中，m 是输出维度（例如，在 MNIST 中 m= 宽度 × 高度 × 通道 =28×28×1=784）。x_i 和 x_i 分别是 x 和 x 的元素。由于损失函数是对输入和输出之间的不相似性的衡量，我们能够使用替代的重建损失函数，如二元交叉熵或结构相似性指数（SSIM）。

与其他神经网络类似，自动编码器在训练过程中试图使这个误差或损失函数尽可能的小。图6.1显示了一个自动编码器。编码器是一个将输入 x 压缩成低维潜伏向量 z 的函数，这个潜伏向量代表输入分布的重要特征。然后，解码器试图以 x 的形式从潜向量中恢复原始输入。

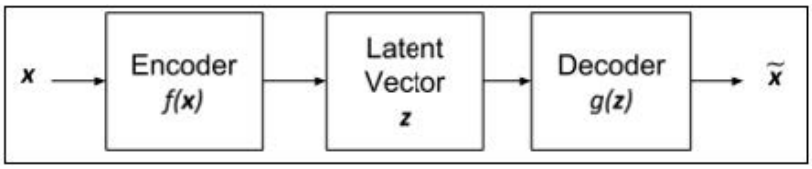

图 6.1: 自动编码器的流程图

为了把自动编码器放在背景中，x 可以是一个 MNIST 数字，其维度为 28×28×1=784。编码器将输入转化为一个低维的 z，可以是一个 16 维的潜在向量。解码器将试图以 x 的形式从 z 恢复输入。

从视觉上看，每个 MNIST 数字 x 看起来都与 x 相似。图**??**向我们展示了这个自动编码过程。

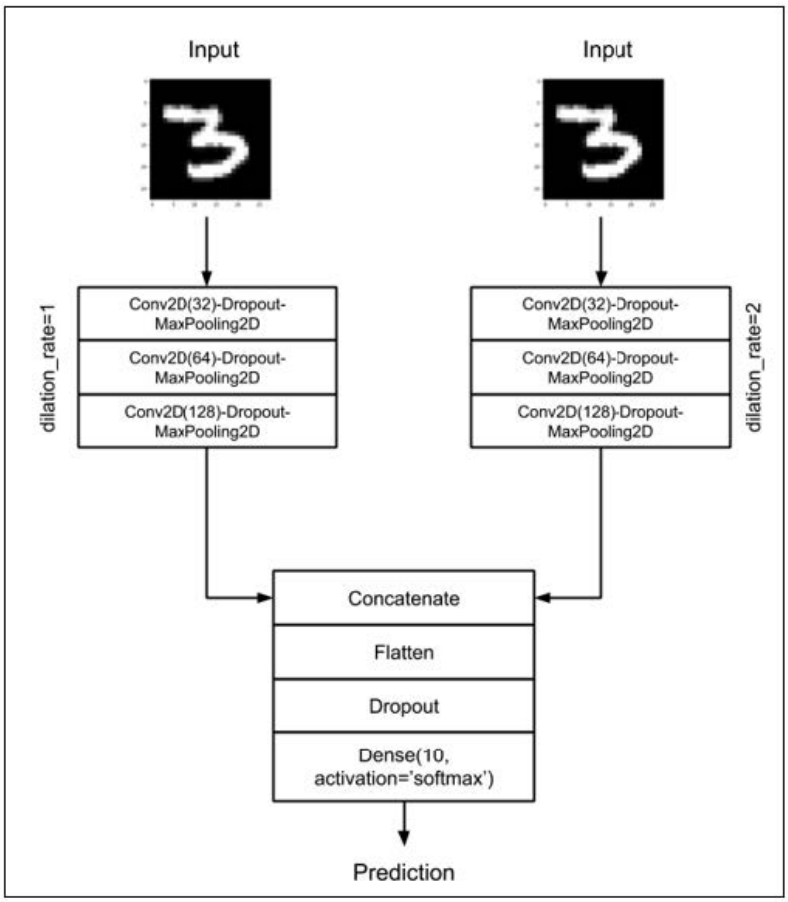

图 **6.2:** An autoencoder with MNIST digit input and output. The latent vector is 16-dim

我们可以观察到，解码后的数字 7 虽然不完全相同，但仍然足够接近。由于编码器和解码器都是非线性函数，我们可以使用神经网络来实现这两者。例如，在 MNIST 数据集中，自动编码器可以由 MLP 或 CNN 实现。自动编码器可以通过反向传播使损失函数最小化来训练。与其他神经网络类似，反向传播的一个要求是损失函数必须是可分的。如果我们把输入当作一个分布，我们可以把编码器解释为分布的编码器，$p(z|x)$，而把解码器解释为分布的解码器，$p(x|z)$。自动编码器的损失函数表示如下。

$$L = -\log p(x|z) \tag{6.2}$$

损失函数只是意味着我们希望在给定潜伏矢量分布的情况下，恢复输入分布的机会最大化。如果假设解码器的输出分布是高斯的，那么损失函数就归结为 MSE，因为。

$$L = -\log p(x|z) = -\log \prod_{i=1}^{m} \mathcal{N}(x_i; x_i, \sigma^2) = -\sum_{i=1} m \log \mathcal{N}(x_i; x_i, \sigma^2) \square \propto_{i=1}^{m} (x_i - x_i)^2 \tag{6.3}$$

在这个例子中，$\mathcal{N}(x_i; x_i, \sigma^2)$ 代表高斯分布，其平均值为 x_i，方差为 σ^2。假设方差为常数。解码器的输出 x_i 被认为是独立的，m 是输出尺寸。了解自动编码器背后的原理将有助于我们的代码实现。在下一节，我们将看看如何使用 tf.keras 功能 API 来构建编码器、解码器和自动编码器。

6.2. 使用 Keras 构建一个自动编码器

使用 tf.keras 库建立一个自动编码器，为了简单起见，将使用 MNIST 数据集作为第一组例子。然后，自动编码器将从输入数据中生成一个潜向量，并使用解码器恢复输入。这第一个例子中的潜伏向量是 16-dim。

首先，我们要通过建立编码器来实现自动编码器。

程序 3.2.1 显示了将 MNIST 数字压缩成一个 16 维潜伏向量的编码器。该编码器是两个 Conv2D 的堆叠。最后阶段是一个有 16 个单元的 Dense 层，用来生成潜向量。

Listing 6.1: Listing 3.2.1: autoencoder-mnist-3.2.1.py

```
Listing 3.2.1: autoencoder-mnist-3.2.1.py
   from tensorflow.keras.layers import Dense, Input
   from tensorflow.keras.layers import Conv2D, Flatten
   from tensorflow.keras.layers import Reshape, Conv2DTranspose
   from tensorflow.keras.models import Model
   from tensorflow.keras.datasets import mnist
   from tensorflow.keras.utils import plot_model
   from tensorflow.keras import backend as K
   import numpy as np
   import matplotlib.pyplot as plt
   # load MNIST dataset
   (x_train, _), (x_test, _) = mnist.load_data()
   # reshape to (28, 28, 1) and normalize input images
   image_size = x_train.shape[1]
   x_train = np.reshape(x_train, [-1, image_size, image_size, 1])
   x_test = np.reshape(x_test, [-1, image_size, image_size, 1])
   x_train = x_train.astype('float32') / 255
   x_test = x_test.astype('float32') / 255
   # network parameters
   input_shape = (image_size, image_size, 1)
   batch_size = 32
   kernel_size = 3
   latent_dim = 16
   # encoder/decoder number of CNN layers and filters per layer
   layer_filters = [32, 64]
   # build the autoencoder model
   # first build the encoder model
   inputs = Input(shape=input_shape, name='encoder_input')
   x = inputs
   # stack of Conv2D(32)-Conv2D(64)
   for filters in layer_filters:
       x = Conv2D(filters=filters,
                  kernel_size=kernel_size,
                  activation='relu',
                  strides=2,
                  padding='same')(x)
   # shape info needed to build decoder model
   # so we don't do hand computation
   # the input to the decoder's first
   # Conv2DTranspose will have this shape
   # shape is (7, 7, 64) which is processed by
   # the decoder back to (28, 28, 1)
   shape = K.int_shape(x)
   # generate latent vector
   x = Flatten()(x)
```

```
    latent = Dense(latent_dim, name='latent_vector')(x)
    # instantiate encoder model
    encoder = Model(inputs,
                    latent,
                    name='encoder')
    encoder.summary()
    plot_model(encoder,
               to_file='encoder.png',
               show_shapes=True)
    # build the decoder model
    latent_inputs = Input(shape=(latent_dim,), name='decoder_input')
    # use the shape (7, 7, 64) that was earlier saved
    x = Dense(shape[1] * shape[2] * shape[3])(latent_inputs)
    # from vector to suitable shape for transposed conv
    x = Reshape((shape[1], shape[2], shape[3]))(x)
    # stack of Conv2DTranspose(64)-Conv2DTranspose(32)
    for filters in layer_filters[::-1]:
        x = Conv2DTranspose(filters=filters,
                            kernel_size=kernel_size,
                          activation='relu',
                          strides=2,
                          padding='same')(x)
# reconstruct the input
outputs = Conv2DTranspose(filters=1,
                          kernel_size=kernel_size,
                          activation='sigmoid',
                          padding='same',
                          name='decoder_output')(x)
# instantiate decoder model
decoder = Model(latent_inputs, outputs, name='decoder')
decoder.summary()
plot_model(decoder, to_file='decoder.png', show_shapes=True)
# autoencoder = encoder + decoder
# instantiate autoencoder model
autoencoder = Model(inputs,
                    decoder(encoder(inputs)),
                    name='autoencoder')
autoencoder.summary()
plot_model(autoencoder,
           to_file='autoencoder.png',
           show_shapes=True)
# Mean Square Error (MSE) loss function, Adam optimizer
autoencoder.compile(loss='mse', optimizer='adam')
# train the autoencoder
autoencoder.fit(x_train,
                x_train,
                validation_data=(x_test, x_test),
                epochs=1,
                batch_size=batch_size)
# predict the autoencoder output from test data
x_decoded = autoencoder.predict(x_test)
# display the 1st 8 test input and decoded images
imgs = np.concatenate([x_test[:8], x_decoded[:8]])
imgs = imgs.reshape((4, 4, image_size, image_size))
imgs = np.vstack([np.hstack(i) for i in imgs])
plt.figure()
   plt.axis('off')
```

```
    plt.title('Input: 1st 2 rows, Decoded: last 2 rows')
    plt.imshow(imgs, interpolation='none', cmap='gray')
    plt.savefig('input_and_decoded.png')
    plt.show()
```

图6.3显示了由 plot_model() 生成的架构模型图，它与 encoder.summary() 生成的文本版本相同。最后一个 Conv2D 输出的形状被保存下来，用于计算解码器输入层的尺寸，以便于重建 MNIST 图像：shape = K.int_shape(x).

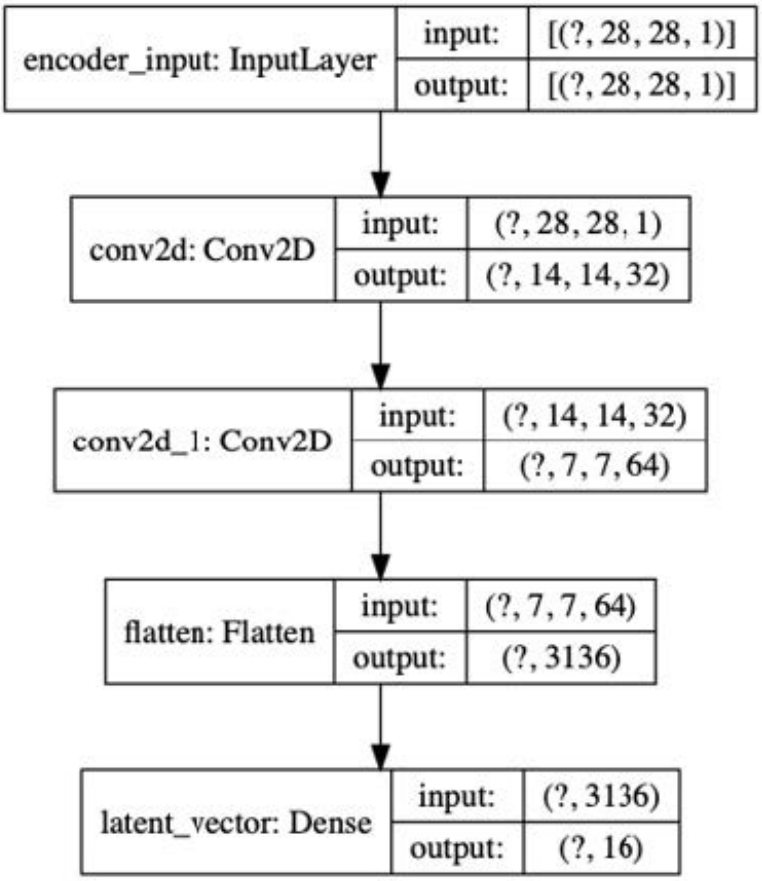

图 6.3: 编码器模型由 Conv2D(32)-Conv2D(64)-Dense(16) 组成，以产生低维潜伏向量。

程序 3.2.1 中的解码器对潜伏向量进行解压，以恢复 MNIST 的数字。解码器的输入阶段是一个密集层，将接受潜伏向量。单位的数量等于编码器保存的 Conv2D 输出尺寸的乘积。这样做是为了让我们可以很容易地调整 Dense 层的输出尺寸，以便 Conv2DTranspose 最终恢复原始 MNIST 图像尺寸。

解码器是由三个 Conv2DTranspose 堆叠而成的。在我们的案例中，我们要使用一个转置的 CNN（有时称为解卷积），这在解码器中比较常用。我们可以把转置的 CNN（Conv2DTranspose）想象成 CNN 的反转过程。

在一个简单的例子中，如果 CNN 将图像转换为特征图，那么转置的 CNN 将产生一个给定特征图的图像。图6.4显示了解码器的模型。

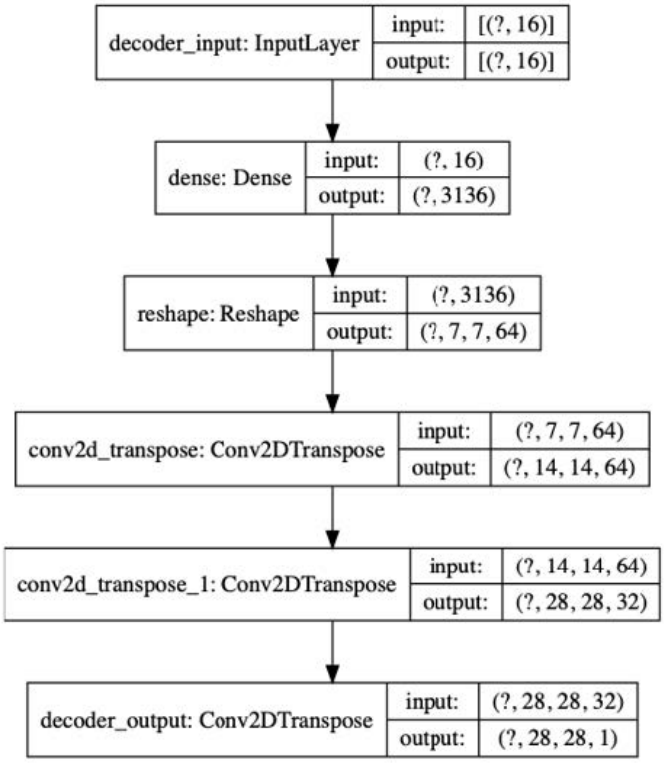

图 6.4: 解码器模型由 Dense(16)-Conv2DTranspose(64)-Conv2DTranspose(32)-Conv2DTranspose（1）组成。输入是被解码的潜伏向量，以恢复原始输入.

通过将编码器和解码器连接在一起，我们就能建立自动编码器。图6.5说明了自动编码器的模型图。

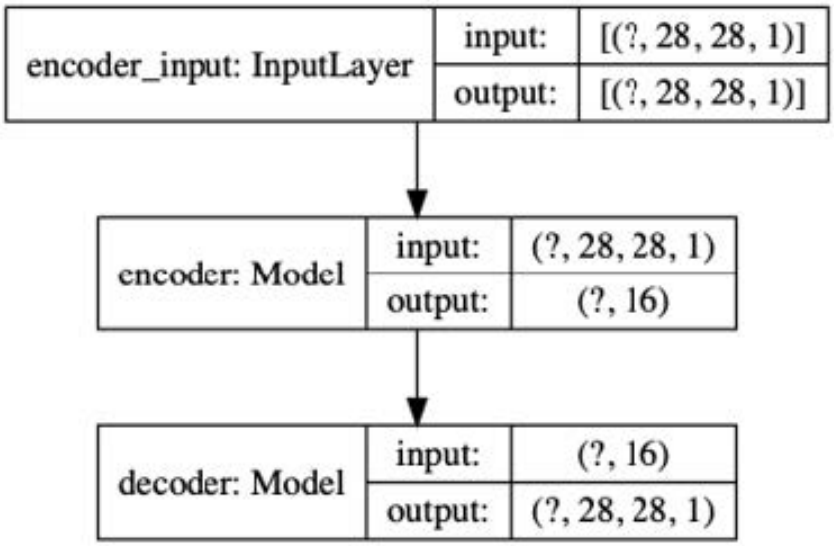

图 6.5: 自动编码器模型是由一个编码器模型和一个解码器模型联合起来建立的。这个自动编码器有 178 个 k 参数.

编码器的张量输出也是生成自动编码器输出的解码器的输入。在这个例子中，我们将使用 MSE 损失函数和亚当优化器。在训练期间，输入与输出相同，即 x_train。我们应该注意到，在我们的例子中，只有几个层，足以在一个 epoch 内将验证损失驱动到 0.01。对于更复杂的数据集，我们可能需要一个更深的编码器和解码器，以及更多的历时训练。

在用 0.01 的验证损失训练自动编码器一个历时后，我们就能验证它是否能编码和解码它以前没有见过的 MNIST 数据。图6.2向我们展示了测试数据的八个样本和相应的解码图像。

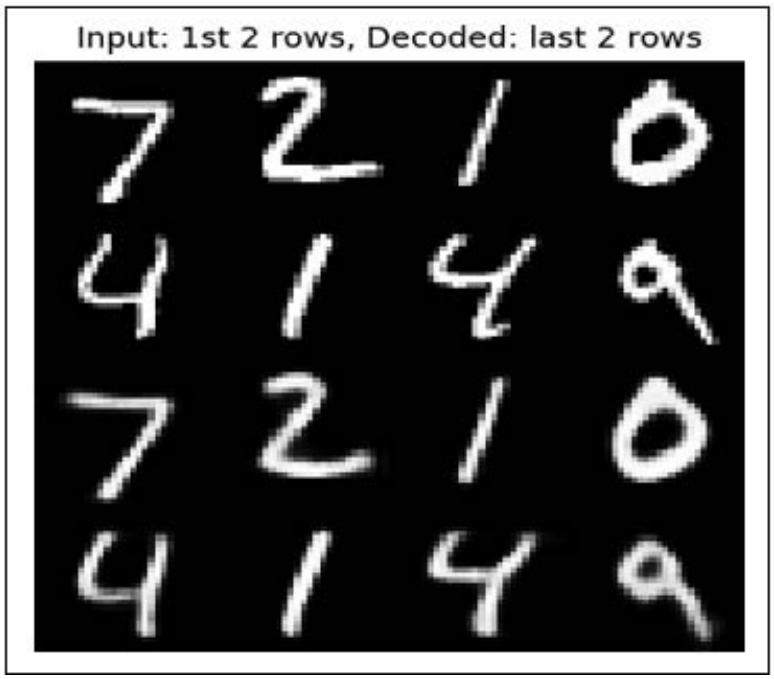

图 6.6: 根据测试数据对自动编码器进行预测。前两行是原始输入测试数据。最后两行是预测的数据。

除了图像的轻微模糊，我们能够很容易地认识到自动编码器能够以良好的质量恢复输入。随着我们对更多的历时进行训练，结果将得到改善。在这一点上，我们可能会想：我们如何在空间中可视化潜伏向量？一个简单的可视化方法是强迫自动编码器使用二维潜伏向量来学习 MNIST 数字特征。从那里，我们能够将这个潜伏向量投射到一个二维空间上，以便看到 MNIST 潜伏向量是如何分布的。图 3.2.5 和图 3.2.6 显示了 MNIST 数字的分布与潜伏码尺寸的关系。

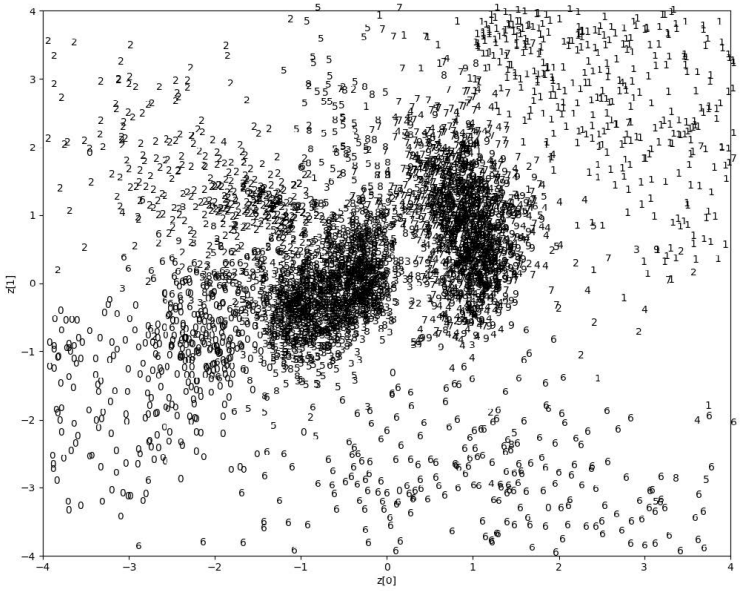

图 6.7: MNIST 的数字分布是潜伏代码维度 z[0] 和 z[1] 的函数。

在图6.7中，我们可以看到特定数字的潜伏向量在空间中的某个区域聚集。例如，数字 0 在左下象限，而数字 1 则在右上象限。这种聚类在图中得到了反映。事实上，同一个图显示了从潜伏空间导航或生成新数字的结果，如图 3.2.5 所示。

例如，从中心开始，朝着右上象限改变一个 2-dim 潜伏向量的值，这让我们看到数字从 9 变成了 1。这是预期的，因为从图 3.2.5 中，我们能够看到数字 9 的潜伏码值聚集在中心附近，而数字 1 的码值聚集在右上象限。

对于图6.7和图6.8，我们只探索了每个潜在向量维度的-4.0 和 +4.0 之间的区域。

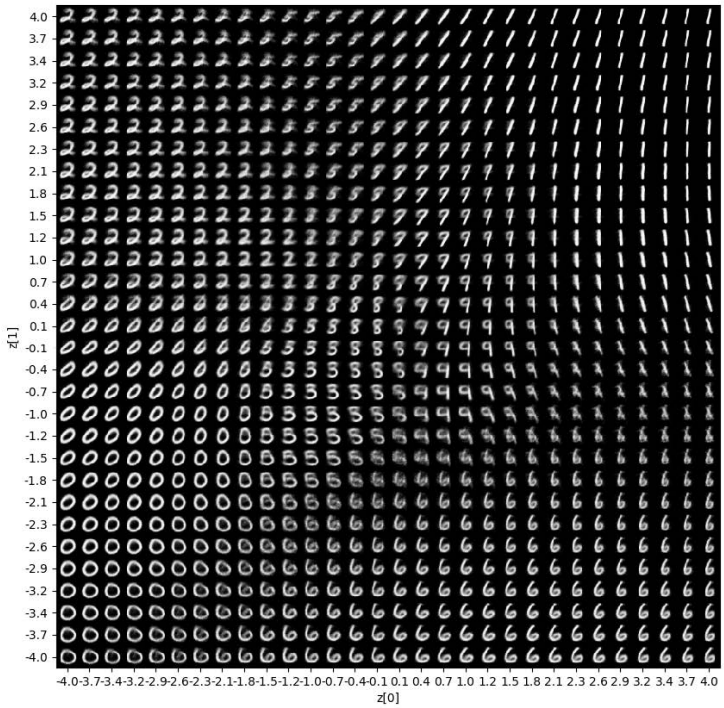

图 6.8: 在二维潜伏向量空间导航时产生的数字

从图6.7中可以看出，潜伏代码分布不是连续的。理想情况下，它应该像一个圆形，到处都有有效的数值。由于这种不连续性，在有些区域，如果我们对潜伏矢量进行解码，几乎不会产生任何可识别的数字。

图6.7和图6.8是在训练了 20 个历时后产生的。autoencoder-mnist-3.2.1.py 的代码经过修改，设置 latent_dim = 2。plot_ results() 函数将 MNIST 数字作为 2-dim 潜伏向量的函数绘制出来。为了方便起见，该程序被保存为 autoencoder-2dim- mnist-3.2.2.py，部分代码见程序 3.2.2。其余的代码实际上与程序 3.2.1 相似，这里不再显示。

Listing 6.2: Listing 3.2.2: autoencoder-2dim-mnist-3.2.2.py

```
Listing 3.2.2: autoencoder-2dim-mnist-3.2.2.py
   def plot_results(models,
                    data,
                    batch_size=32,
             model_name="autoencoder_2dim"):
"""Plots 2-dim latent values as scatter plot of digits
    then, plot MNIST digits as function of 2-dim latent vector
Arguments:
    models (list): encoder and decoder models
    data (list): test data and label
    batch_size (int): prediction batch size
    model_name (string): which model is using this function
"""
encoder, decoder = models
x_test, y_test = data
xmin = ymin = -4
```

```
xmax = ymax = +4
os.makedirs(model_name, exist_ok=True)
filename = os.path.join(model_name, "latent_2dim.png")
# display a 2D plot of the digit classes in the latent space
z = encoder.predict(x_test,
                    batch_size=batch_size)
plt.figure(figsize=(12, 10))
# axes x and y ranges
axes = plt.gca()
axes.set_xlim([xmin,xmax])
axes.set_ylim([ymin,ymax])
# subsample to reduce density of points on the plot
z = z[0::2]
y_test = y_test[0::2]
plt.scatter(z[:, 0], z[:, 1], marker="")
for i, digit in enumerate(y_test):
    axes.annotate(digit, (z[i, 0], z[i, 1]))
plt.xlabel("z[0]")
plt.ylabel("z[1]")
plt.savefig(filename)
plt.show()
filename = os.path.join(model_name, "digits_over_latent.png")
# display a 30x30 2D manifold of the digits
n = 30
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))
       # linearly spaced coordinates corresponding to the 2D plot
       # of digit classes in the latent space
       grid_x = np.linspace(xmin, xmax, n)
       grid_y = np.linspace(ymin, ymax, n)[::-1]
       for i, yi in enumerate(grid_y):
           for j, xi in enumerate(grid_x):
               z = np.array([[xi, yi]])
               x_decoded = decoder.predict(z)
               digit = x_decoded[0].reshape(digit_size, digit_size)
               figure[i * digit_size: (i + 1) * digit_size,
                      j * digit_size: (j + 1) * digit_size] = digit
       plt.figure(figsize=(10, 10))
       start_range = digit_size // 2
       end_range = n * digit_size + start_range + 1
       pixel_range = np.arange(start_range, end_range, digit_size)
       sample_range_x = np.round(grid_x, 1)
       sample_range_y = np.round(grid_y, 1)
       plt.xticks(pixel_range, sample_range_x)
       plt.yticks(pixel_range, sample_range_y)
       plt.xlabel("z[0]")
       plt.ylabel("z[1]")
       plt.imshow(figure, cmap='Greys_r')
       plt.savefig(filename)
       plt.show()
```

这就完成了对自动编码器的实现和研究。接下来的几章将集中讨论它们的实际应用。我们将从去噪自动编码器开始。

6.3. 去噪自动编码器 (DAE)

我们现在要建立一个有实际应用的自动编码器。首先，让我们描绘一下，想象一下 MNIST 数字图像被噪声所破坏，从而使人类更难阅读。我们能够建立一个去噪自动编码器（DAE）来去除这些图像中的噪声。图6.9向我们展示了三组 MNIST 的数字。每组的顶行（例如，MNIST 数字 7、2、1、9、0、6、3、4 和 9）是原始图像。中间几行是 DAE 的输入，是被噪声破坏的原始图像。作为人类，可以发现，阅读被破坏的 MNIST 数字是很困难的。最后几行是 DAE 的输出。

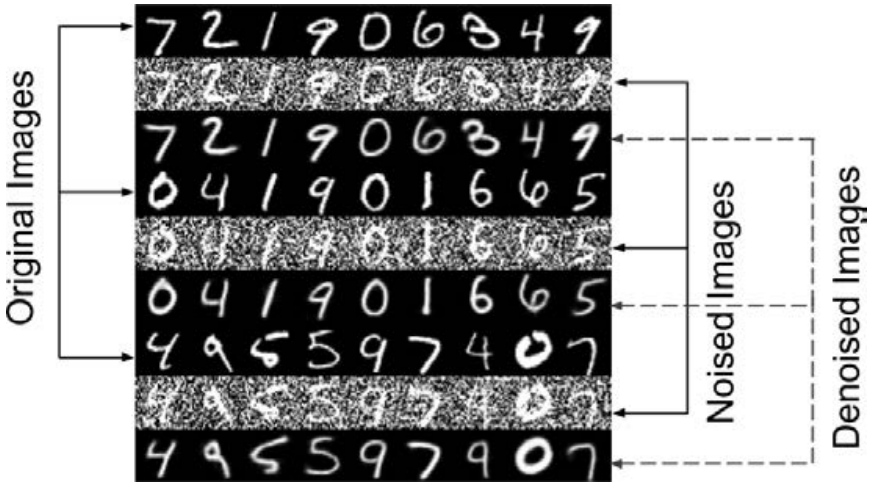

图 6.9: 原始 MNIST 数字（最上面几行），损坏的原始图像（中间几行）和去噪图像（最后几行）。

如图6.10所示，去噪自动编码器与我们在上一节介绍的 MNIST 自动编码器的结构基本相同。

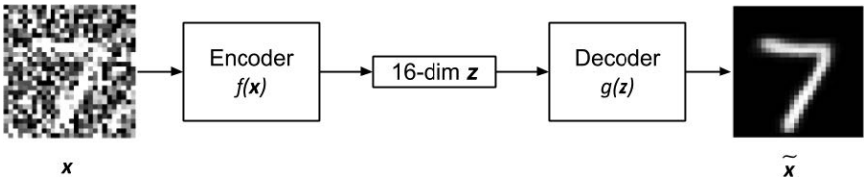

图 6.10: 去噪自动编码器的输入是被破坏的图像。输出是干净或去噪的图像。潜伏向量被认为是 16 个维度的

图 3.3.2 中的输入被定义为。

$$x = x_{orig} + nosie \tag{6.4}$$

在这个公式中，x_{orig} 代表被噪声破坏的原始 MNIST 图像。编码器的目标是发现如何产生潜伏向量 z，这将使解码器能够恢复如 MSE，如通过：x_{orig} 最小化异质性损失函数所示。

$$L = (x_{orig}, x) = MSE + \frac{\sum}{i = 1^{i=m}(x_{orig_i} - x_i)^2} \tag{6.5}$$

在这个例子中，m 是输出尺寸（例如，在 MNIST 中，m= 宽度 × 高度 × 通道 =28×28×1=784）。x_{orig} 和 x_i 分别是 x_{orig} 和 x 的元素。

为了实现 DAE，我们需要对上一节中介绍的自动编码器做一些改变。首先，训练输入数据应该是被破坏的 MNIST 数字。训练输出数据是相同的原始干净的 MNIST 数字。这就好比是告诉自

动编码器纠正后的图像应该是什么，或者要求它找出如何去除给定的损坏图像的噪声。最后，我们必须在被破坏的 MNIST 测试数据上验证自动编码器。

图6.10左边的 MNIST 数字 7 是一个实际的损坏的图像输入。右边的则是经过训练的去噪自动编码器的干净图像输出。

Listing 6.3: Listing 3.3.1: denoising-autoencoder-mnist-3.3.1.py

```
1
2    from tensorflow.keras.layers import Dense, Input
3    from tensorflow.keras.layers import Conv2D, Flatten
4    from tensorflow.keras.layers import Reshape, Conv2DTranspose
5    from tensorflow.keras.models import Model
6    from tensorflow.keras import backend as K
7    from tensorflow.keras.datasets import mnist
8    import numpy as np
9    import matplotlib.pyplot as plt
10   from PIL import Image
11   np.random.seed(1337)
12   # load MNIST dataset
13   (x_train, _), (x_test, _) = mnist.load_data()
14   # reshape to (28, 28, 1) and normalize input images
15   image_size = x_train.shape[1]
16   x_train = np.reshape(x_train, [-1, image_size, image_size, 1])
17   x_test = np.reshape(x_test, [-1, image_size, image_size, 1])
18   x_train = x_train.astype('float32') / 255
19   x_test = x_test.astype('float32') / 255
20   # generate corrupted MNIST images by adding noise with normal dist
21   # centered at 0.5 and std=0.5
22   noise = np.random.normal(loc=0.5, scale=0.5, size=x_train.shape)
23   x_train_noisy = x_train + noise
24   noise = np.random.normal(loc=0.5, scale=0.5, size=x_test.shape)
25   x_test_noisy = x_test + noise
26   # adding noise may exceed normalized pixel values>1.0 or <0.0
27   # clip pixel values >1.0 to 1.0 and <0.0 to 0.0
28   x_train_noisy = np.clip(x_train_noisy, 0., 1.)
29   x_test_noisy = np.clip(x_test_noisy, 0., 1.)
30   # network parameters
31input_shape = (image_size, image_size, 1)
32batch_size = 32
33kernel_size = 3
34latent_dim = 16
35# encoder/decoder number of CNN layers and filters per layer
36layer_filters = [32, 64]
37# build the autoencoder model
38# first build the encoder model
39inputs = Input(shape=input_shape, name='encoder_input')
40x = inputs
41# stack of Conv2D(32)-Conv2D(64)
42for filters in layer_filters:
43    x = Conv2D(filters=filters,
44               kernel_size=kernel_size,
45               strides=2,
46               activation='relu',
47               padding='same')(x)
48# shape info needed to build decoder model so we don't do hand
49computation
50# the input to the decoder's first Conv2DTranspose will have this
```

```
shape
# shape is (7, 7, 64) which can be processed by the decoder back to
(28, 28, 1)
shape = K.int_shape(x)
# generate the latent vector
x = Flatten()(x)
latent = Dense(latent_dim, name='latent_vector')(x)
# instantiate encoder model
encoder = Model(inputs, latent, name='encoder')
encoder.summary()
# build the decoder model
latent_inputs = Input(shape=(latent_dim,), name='decoder_input')
# use the shape (7, 7, 64) that was earlier saved
x = Dense(shape[1] * shape[2] * shape[3])(latent_inputs)
# from vector to suitable shape for transposed conv
x = Reshape((shape[1], shape[2], shape[3]))(x)
# stack of Conv2DTranspose(64)-Conv2DTranspose(32)
for filters in layer_filters[::-1]:
    x = Conv2DTranspose(filters=filters,
                        kernel_size=kernel_size,
                        strides=2,
                        activation='relu',
                        padding='same')(x)
# reconstruct the denoised input
outputs = Conv2DTranspose(filters=1,
                          kernel_size=kernel_size,
                          padding='same',
                          activation='sigmoid',
                          name='decoder_output')(x)
# instantiate decoder model
decoder = Model(latent_inputs, outputs, name='decoder')
decoder.summary()
# autoencoder = encoder + decoder
# instantiate autoencoder model
autoencoder = Model(inputs, decoder(encoder(inputs)),
name='autoencoder')
autoencoder.summary()
# Mean Square Error (MSE) loss function, Adam optimizer
autoencoder.compile(loss='mse', optimizer='adam')
# train the autoencoder
autoencoder.fit(x_train_noisy,
                x_train,
                validation_data=(x_test_noisy, x_test),
                epochs=10,
                batch_size=batch_size)
# predict the autoencoder output from corrupted test images
x_decoded = autoencoder.predict(x_test_noisy)
# 3 sets of images with 9 MNIST digits
# 1st rows - original images
# 2nd rows - images corrupted by noise
# 3rd rows - denoised images
rows, cols = 3, 9
num = rows * cols
imgs = np.concatenate([x_test[:num], x_test_noisy[:num], x_
decoded[:num]])
imgs = imgs.reshape((rows * 3, cols, image_size, image_size))
imgs = np.vstack(np.split(imgs, rows, axis=1))
```

```
imgs = imgs.reshape((rows * 3, -1, image_size, image_size))
imgs = np.vstack([np.hstack(i) for i in imgs])
 imgs = (imgs * 255).astype(np.uint8)
   plt.figure()
   plt.axis('off')
   plt.title('Original images: top rows, '
             'Corrupted Input: middle rows, '
             'Denoised Input:  third rows')
   plt.imshow(imgs, interpolation='none', cmap='gray')
   Image.fromarray(imgs).save('corrupted_and_denoised.png')
   plt.show()
```

程序 3.3.1 显示了去噪自动编码器，它已被贡献到 Keras 的官方 GitHub 仓库。使用相同的 MNIST 数据集，我们能够通过添加随机噪声来模拟损坏的图像。添加的噪声是一个高斯分布，其平均值为 $\mu = 0.5$，标准偏差为 $\sigma = 0.5$。由于添加随机噪声可能会将像素数据推向小于 0 或大于 1 的无效值，所以像素值被剪切到 [0.1, 1.0] 范围。

其他一切都将与上一节中的自动编码器基本保持一致。我们将使用相同的 MSE 损失函数和 Adam 优化器。然而，训练的历时数增加到 10。这是为了允许充分的参数优化。

图6.11显示，当噪声水平从 $\sigma = 0.5$ 增加到 $\sigma = 0.75$ 和 $\sigma = 1.0$ 时，DAE 有一定程度的稳健性。在 $\sigma = 0.75$ 时，DAE 仍然能够恢复原始图像。然而，当 $\sigma = 1.0$ 时，一些数字，如第二和第三组中的 4 和 5，就不能再被正确恢复了。

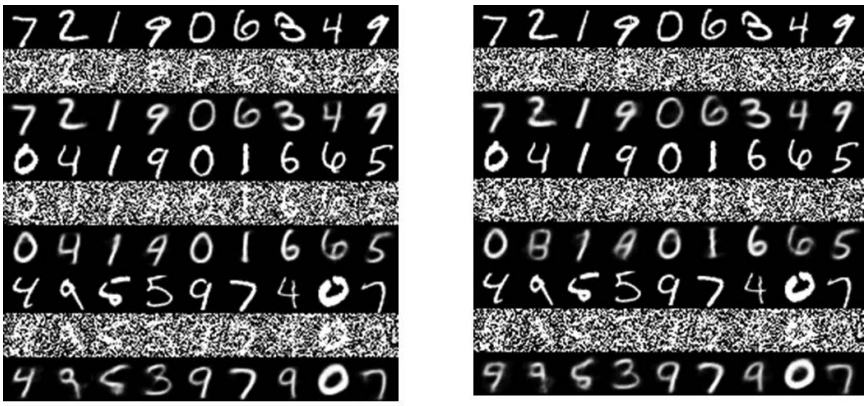

图 6.11: 随着噪声水平的提高，去噪自动编码器的性能。

我们已经完成了去噪自动编码器的讨论和实现。虽然这个概念是在 MNIST 数字上演示的，但这个想法也适用于其他信号。在下一节，我们将介绍自动编码器的另一个实际应用，即着色自动编码器。

6.4. 自动着色的自动编码器

我们现在要做的是自动编码器的另一个实际应用。在这种情况下，我们将想象我们有一张灰度照片，我们想建立一个工具来自动为它添加颜色。我们想复制人类的能力，识别出大海和天空是蓝色的，草场和树木是绿色的，而云是白色的，等等。如图6.12所示，如果给我们一张灰度照片（左），前景是稻田，背景是火山，顶部是天空，我们就能够添加适当的颜色（右）。

图 6.12: 为马荣火山的灰度照片添加颜色。着色网络应该通过向灰度照片添加颜色来复制人类的能力。左边的照片是灰度的。右边的照片是彩色的。

一个简单的自动着色算法似乎是一个适合自动编码器的问题。如果我们能用足够数量的灰度照片作为输入，并将相应的彩色照片作为输出来训练自动编码器，它就有可能发现正确应用颜色的隐藏结构。粗略地说，这是去噪的反向过程。问题是，自动编码器能否在原始灰度图像上添加颜色（良好的噪声）?

程序 3.4.1 显示了着色自动编码器网络。着色自动编码器网络是我们用于 MNIST 数据集的去噪自动编码器的一个修改版。首先，我们需要一个从灰度到彩色照片的数据集。我们之前使用过的 CIFAR10 数据库有 50,000 张训练用和 10,000 张测试用的 32×32 RGB 照片，可以转换为灰阶。如以下列表所示，我们能够使用 rgb2gray() 函数在 R、G 和 B 分量上应用权重，将彩色转换为灰度。

Listing 6.4: Listing 3.4.1: colorization-autoencoder-cifar10-3.4.1.py

```
Listing 3.4.1: colorization-autoencoder-cifar10-3.4.1.py
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.layers import Conv2D, Flatten
from tensorflow.keras.layers import Reshape, Conv2DTranspose
from tensorflow.keras.models import Model
from tensorflow.keras.callbacks import ReduceLROnPlateau
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import plot_model
from tensorflow.keras import backend as K
import numpy as np
import matplotlib.pyplot as plt
import os
def rgb2gray(rgb):
    """Convert from color image (RGB) to grayscale.
       Source: opencv.org
       grayscale = 0.299*red + 0.587*green + 0.114*blue
    Argument:
        rgb (tensor): rgb image
    Return:
        (tensor): grayscale image
    """
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])
```

```
    # load the CIFAR10 data
    (x_train, _), (x_test, _) = cifar10.load_data()
    # input image dimensions
    # we assume data format "channels_last"
    img_rows = x_train.shape[1]
    img_cols = x_train.shape[2]
    channels = x_train.shape[3]
    # create saved_images folder
imgs_dir = 'saved_images'
save_dir = os.path.join(os.getcwd(), imgs_dir)
if not os.path.isdir(save_dir):
        os.makedirs(save_dir)
# display the 1st 100 input images (color and gray)
imgs = x_test[:100]
imgs = imgs.reshape((10, 10, img_rows, img_cols, channels))
imgs = np.vstack([np.hstack(i) for i in imgs])
plt.figure()
plt.axis('off')
plt.title('Test color images (Ground  Truth)')
plt.imshow(imgs, interpolation='none')
plt.savefig('%s/test_color.png' % imgs_dir)
plt.show()
# convert color train and test images to gray
x_train_gray = rgb2gray(x_train)
x_test_gray = rgb2gray(x_test)
# display grayscale version of test images
imgs = x_test_gray[:100]
imgs = imgs.reshape((10, 10, img_rows, img_cols))
imgs = np.vstack([np.hstack(i) for i in imgs])
plt.figure()
plt.axis('off')
plt.title('Test gray images (Input)')
plt.imshow(imgs, interpolation='none', cmap='gray')
plt.savefig('%s/test_gray.png' % imgs_dir)
plt.show()
# normalize output train and test color images
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
# normalize input train and test grayscale images
x_train_gray = x_train_gray.astype('float32') / 255
x_test_gray = x_test_gray.astype('float32') / 255
# reshape images to row x col x channel for CNN output/validation
x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols,
channels)
x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, channels)
# reshape images to row x col x channel for CNN input
x_train_gray = x_train_gray.reshape(x_train_gray.shape[0], img_rows,
img_cols, 1)
x_test_gray = x_test_gray.reshape(x_test_gray.shape[0], img_rows, img_
cols, 1)
# network parameters
input_shape = (img_rows, img_cols, 1)
batch_size = 32
kernel_size = 3
latent_dim = 256
# encoder/decoder number of CNN layers and filters per layer
layer_filters = [64, 128, 256]
```

```
# build the autoencoder model
# first build the encoder model
inputs = Input(shape=input_shape, name='encoder_input')
x = inputs
# stack of Conv2D(64)-Conv2D(128)-Conv2D(256)
for filters in layer_filters:
    x = Conv2D(filters=filters,
               kernel_size=kernel_size,
               strides=2,
               activation='relu',
               padding='same')(x)
# shape info needed to build decoder model so we don't do hand
computation
# the input to the decoder's first Conv2DTranspose will have this
shape
# shape is (4, 4, 256) which is processed by the decoder back to (32,
32, 3)
shape = K.int_shape(x)
# generate a latent vector
x = Flatten()(x)
latent = Dense(latent_dim, name='latent_vector')(x)
# instantiate encoder model
encoder = Model(inputs, latent, name='encoder')
encoder.summary()
  # build the decoder model
   latent_inputs = Input(shape=(latent_dim,), name='decoder_input')
   x = Dense(shape[1]*shape[2]*shape[3])(latent_inputs)
   x = Reshape((shape[1], shape[2], shape[3]))(x)
   # stack of Conv2DTranspose(256)-Conv2DTranspose(128)-
   Conv2DTranspose(64)
   for filters in layer_filters[::-1]:
       x = Conv2DTranspose(filters=filters,
                           kernel_size=kernel_size,
                           strides=2,
                           activation='relu',
                           padding='same')(x)
   outputs = Conv2DTranspose(filters=channels,
                             kernel_size=kernel_size,
                             activation='sigmoid',
                             padding='same',
                             name='decoder_output')(x)
   # instantiate decoder model
   decoder = Model(latent_inputs, outputs, name='decoder')
   decoder.summary()
   # autoencoder = encoder + decoder
   # instantiate autoencoder model
   autoencoder = Model(inputs, decoder(encoder(inputs)),
   name='autoencoder')
   autoencoder.summary()
   # prepare model saving directory.
   save_dir = os.path.join(os.getcwd(), 'saved_models')
   model_name = 'colorized_ae_model.{epoch:03d}.h5'
   if not os.path.isdir(save_dir):
           os.makedirs(save_dir)
   filepath = os.path.join(save_dir, model_name)
   # reduce learning rate by sqrt(0.1) if the loss does not improve in 5
   epochs
```

```
    lr_reducer = ReduceLROnPlateau(factor=np.sqrt(0.1),
                                   cooldown=0,
                                   patience=5,
                                   verbose=1,
                                   min_lr=0.5e-6)
    # save weights for future use (e.g. reload parameters w/o training)
    checkpoint = ModelCheckpoint(filepath=filepath,
                                 monitor='val_loss',
                                 verbose=1,
                                 save_best_only=True)
    # Mean Square Error (MSE) loss function, Adam optimizer
    autoencoder.compile(loss='mse', optimizer='adam')
    # called every epoch
    callbacks = [lr_reducer, checkpoint]
    # train the autoencoder
    autoencoder.fit(x_train_gray,
                    x_train,
                    validation_data=(x_test_gray, x_test),
                    epochs=30,
                    batch_size=batch_size,
                    callbacks=callbacks)
    # predict the autoencoder output from test data
    x_decoded = autoencoder.predict(x_test_gray)
    # display the 1st 100 colorized images
    imgs = x_decoded[:100]
    imgs = imgs.reshape((10, 10, img_rows, img_cols, channels))
    imgs = np.vstack([np.hstack(i) for i in imgs])
    plt.figure()
    plt.axis('off')
    plt.title('Colorized test images (Predicted)')
    plt.imshow(imgs, interpolation='none')
    plt.savefig('%s/colorized.png' % imgs_dir)
    plt.show()
```

我们通过增加一个卷积和转置卷积的块来增加自动编码器的容量。我们还将每个 CNN 块的过滤器的数量增加了一倍。潜伏向量现在是 256-dim，以便增加它能代表的突出属性的数量，这在自动编码器部分已经讨论过了。最后，输出滤波器的大小增加到了三个，或者说等于预期彩色输出的 RGB 通道数。

现在用灰度作为输入，原始 RGB 图像作为输出来训练着色自动编码器。训练将花费更多的历时，当验证损失没有改善时，使用学习率降低器来缩小学习率。这可以通过告诉 tf.keras fit() 函数中的 callbacks 参数来调用 lr_reducer() 函数来轻松实现。

图6.13展示了 CIFAR10 的测试数据集中灰度图像的着色。

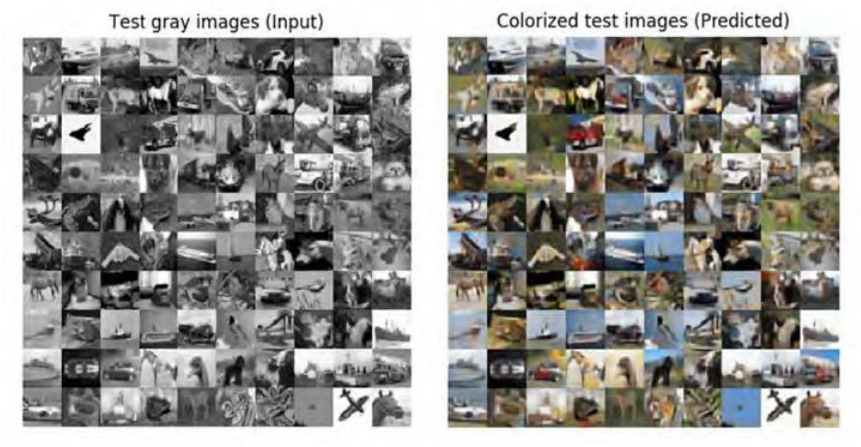

图 6.13: 使用自动编码器进行灰度到彩色图像的自动转换。CIFAR10 测试灰度输入图像（左）和预测的彩色图像（右）。

图6.14比较了地面真相和着色自动编码器的预测。

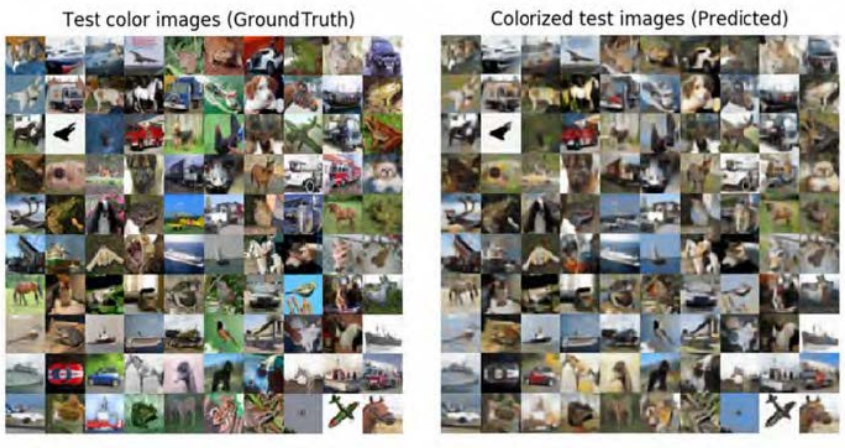

图 6.14: 地面真实的彩色图片和预测的彩色化图片的并排比较

自动编码器完成了一个可接受的着色工作。大海或天空被预测为蓝色，动物有深浅不一的棕色，云是白色的，诸如此类。有一些明显的不正确预测，如红色车辆变成了蓝色，或蓝色车辆变成了红色，偶尔有绿地被误认为是蓝天，黑暗或金色的天空被转换成了蓝天。这是关于自动编码器的最后一节。在接下来的章节中，我们将以这样或那样的形式重新审视编码和解码的概念。表征学习的概念在深度学习中是非常基本的。

6.5. 总结

在本章中，我们已经介绍了自动编码器，它是将输入数据压缩成低维表示的神经网络，以便有效地进行结构转换，如去噪和着色。我们已经为更高级的 GANs 和 VAEs 课题奠定了基础，我们将在后面的章节中介绍这些课题。我们已经演示了如何从两个构件模型中实现一个自动编码器，包括编码器和解码器。我们还学习了如何提取输入分布的隐性结构是人工智能中的常见任务之一。一旦学会了潜伏代码，就可以对原始输入分布进行许多结构性操作。为了更好地理解输入分布，潜

伏向量形式的隐性结构可以使用低级别的嵌入，类似于我们在本章所做的，或者通过更复杂的降维技术，如 t-SNE 或 PCA，来实现可视化。除了去噪和着色，自动编码器还被用于将输入分布转换为低维潜伏向量，这些潜伏向量可以被进一步处理，用于其他任务，如分割、检测、跟踪、重建和视觉理解。在第八章，变异自动编码器（VAEs）中，我们将讨论 VAEs，它在结构上与自动编码器相同，但不同的是它有可解释的潜伏代码，可以产生一个连续的潜伏向量投影。

6.6. 参考文献

1. Ian Goodfellow et al.: Deep Learning. Vol. 1. Cambridge: MIT press, 2016