10

分离表征 GANs

正如我们所探讨的，GANs 可以通过学习数据分布来产生有意义的输出。然而，对生成的输出的属性没有控制。GAN 的一些变体，如前两章所讨论的条件 GAN（CGAN）和辅助分类器 GAN（ACGAN），能够训练一个有条件合成特定输出的发生器。例如，CGAN 和 ACGAN 都能诱导生成器产生一个特定的 MNIST 数字。这是通过使用一个 100 分的噪声代码和相应的一热标签作为输入实现的。然而，除了一热标签之外，我们没有其他方法来控制生成的输出的属性。

在本章中，我们将介绍 GANs 的变化，使我们能够修改生成器的输出。在 MNIST 数据集的背景下，除了要产生哪个数字外，我们可能会发现我们想控制书写风格。这可能涉及所需数字的倾斜度或宽度。换句话说，GANs 也可以学习分解的潜在代码或表示，我们可以用它来改变生成器输出的属性。分解码或表征是一个张量，可以改变输出数据的特定特征或属性，而不影响其他属性。

在本章的第一节，我们将讨论 InfoGAN：通过信息最大化生成对抗网的可解释性表示学习 [1]，这是 GAN 的一个扩展。InfoGAN 通过最大化输入代码和输出观察之间的相互信息，以无监督的方式学习分解的表征。在 MNIST 数据集上，InfoGAN 从数字数据集中分离出了书写风格。在本章的下面部分，我们还将讨论堆叠生成对抗网络或堆叠 GAN[2]，这是 GAN 的另一个扩展。StackedGAN 使用一个预先训练好的编码器或分类器，以帮助分解潜伏代码。StackedGAN 可以被看作是一个模型的堆叠，每个模型都由一个编码器和一个 GAN 组成。每个 GAN 都是通过使用相应编码器的输入和输出数据，以对抗性的方式进行训练。

综上所述，本章的目标是提出。

- 分离表征的概念
- InfoGAN 和 StackedGAN 的原理
- 使用 tf.keras 实现 InfoGAN 和 StackedGAN。

让我们先看一下分离表征的概念。

10.1. 分离表征的概念

最初的 GAN 能够产生有意义的输出，但缺点是其属性无法控制。例如，如果我们训练一个 GAN 来学习名人面孔的分布，生成器就会产生名人模样的新图像。然而，对于我们想要的脸部的

具体属性，没有办法影响生成器。例如，我们无法要求生成器生成一张黑色长发、肤色白皙、棕色眼睛、面带微笑的女性名人的脸。其根本原因是，我们使用的 100-dim 噪声代码纠缠了生成器输出的所有突出属性。我们可以回顾一下，在 tf.keras 中，100-dim 代码是由均匀噪声分布的随机采样产生的。

Listing 10.1: Evaluate how the model does on the test set

```
            # generate fake images from noise using generator
            # generate noise using uniform distribution
            noise = np.random.uniform(-1.0,
                                      1.0,
                                      size=[batch_size, latent_size])
            # generate fake images
            fake_images = generator.predict(noise)
```

如果我们能够修改原始的 GAN，使表示被分离成纠缠的和不纠缠的可解释的潜伏代码向量，我们将能够告诉生成器要合成什么。

图10.1向我们展示了一个带有纠缠代码的 GAN，以及其带有纠缠和非纠缠表示的混合物的变化。在假设的名人脸部生成的背景下，通过纠缠码，我们能够指出我们希望生成的脸部的性别、发型、面部表情、肤色和眼睛颜色。我们仍然需要 n-dim 纠缠码来表示所有其他我们没有纠缠的面部属性，如脸型、面部毛发、眼镜等，这只是三个例子。纠缠码和解缠码向量的连接作为发生器的新输入。串联后的代码的总尺寸不一定是 100。

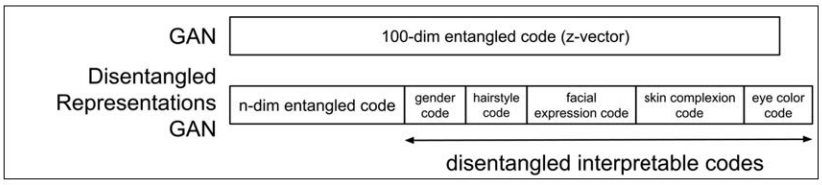

图 10.1: 带有纠缠码的 GAN 及其同时带有纠缠码和非纠缠码的变体。这个例子是在名人面孔生成的背景下显示的

从上图来看，具有分解表征的 GAN 似乎也可以用与 vanilla GAN 相同的方式进行优化。这是因为发生器的输出可以表示为:

$$g(z, c) = g(z) \tag{10.1}$$

代码包含 $z = (z, c)$ 两个元素。

- 类似于 GANsz 或噪声矢量的不可压缩的纠缠噪声代码;
- 潜伏代码，$c_1, c_2, \ldots, c_L$，代表数据分布的可解释的纠缠代码。总的来说，所有潜伏代码都由 c 表示。

为简单起见，假设所有的潜伏代码都是独立的, 如下公式所示:

$$p(c_1, c_2, \cdots, c_L) = \prod_{i=1}^{L} p(c_i) \tag{10.2}$$

生成器函数 $x = g(z, c) = g(z)$ 既有不可压缩噪声码又有潜伏码。从发生器的角度来看，优化 $z = (z, c)$ 和优化 z 是一样的。

生成器网络在提出解决方案时，将简单地忽略分解码所带来的约束。生成器学习分布 $p_g(x|c) = p_g(x)$。这实际上会使分解表征的目标失败。InfoGAN 的关键思想是迫使 GAN 不忽视潜伏代码 c，这是通过最大化 c 和 $g(z,c)$ 之间的相互信息实现的。在下一节，我们将制定 InfoGAN 的损失函数。

10.2. InfoGAN

为了强制执行代码的离散性，InfoGAN 对原始损失函数提出了一个正则器，使潜伏代码 c 和 $g(z,c)$ 之间的互信息最大化，如下公式所示：

$$I(c; g(z,c)) = i(c; g(z)) \tag{10.3}$$

正则器迫使生成器在制定合成假图像的函数时考虑潜伏代码。在信息论领域，潜伏码 c 和 $g(z,c)$ 之间的相互信息定义为：

$$I(c; g(z,c)) = H(c) - H(c|g(z,c)) \tag{10.4}$$

其中 $H(c)$ 是潜伏代码 c 的熵，$H(c|g(z,c))$ 是观察发生器的输出后 c 的条件熵 $g(z,c)$。熵是对随机变量或事件的不确定性的一种衡量。例如，太阳从东方升起这样的信息具有低熵，而在彩票中赢得大奖则具有高熵。关于互信息的更详细的讨论可以在第 13 章，使用互信息的无监督学习中找到。

在公式10.4中，最大化互信息意味着最小化 $H(c|g(z,c))$ 或减少观察到生成输出时的潜代码的不确定性。这是有道理的，因为例如在 MNIST 数据集中，如果 GAN 看到它观察到数字 8，生成器就会对合成数字 8 更有信心。

然而，很难估计 $H(c|g(z,c))$，因为它需要了解第六章后验 $P(c|g(z,c)) = P(c|x)$ ，这是我们无法得到的东西。为了简单起见，我们将使用普通字母 x 来表示数据分布。

解决办法是通过用辅助分布 $Q(c|x)$ 来估计后验的相互信息的下限,InfoGAN 估计相互信息的下限表示为：

$$I(c; g(z,c)) \geq L_I(g,Q) = E_{c\ p(c),x\ g(z,c)}\left[\log Q(c|x)\right] + H(c) \tag{10.5}$$

在 InfoGAN 中，$H(c)$ 被认为是一个常数。因此，最大化相互信息是一个最大化期望值的问题。生成器必须确信它已经生成了一个具有特定属性的输出。我们应该注意，这个期望值的最大值是零。因此，相互信息的下限的最大值是 $H(c)$。在 InfoGAN 中，离散潜伏代码的 $Q(c|x)$ 可以用 softmax 非线性表示。其期望值是 tf.keras 中的负分类交叉熵损失 categorical_crossentropy 。

对于单维的连续编码，期望值是对 c 和 x 的双积分。这是由于期望值来自于离散的编码分布和生成器分布。估计期望值的一种方法是假设样本是连续数据的良好措施。因此，损失估计为 $c\log Q(c|x)$。

为了完成 InfoGAN 的网络，我们应该有一个 $Q(c|x)$ 的实现。为了简单起见，网络 Q 是一个附属于判别器第二至最后一层的辅助网络。因此，这对原始 GAN 的训练影响很小。

图10.2是 InfoGAN 的示意图。

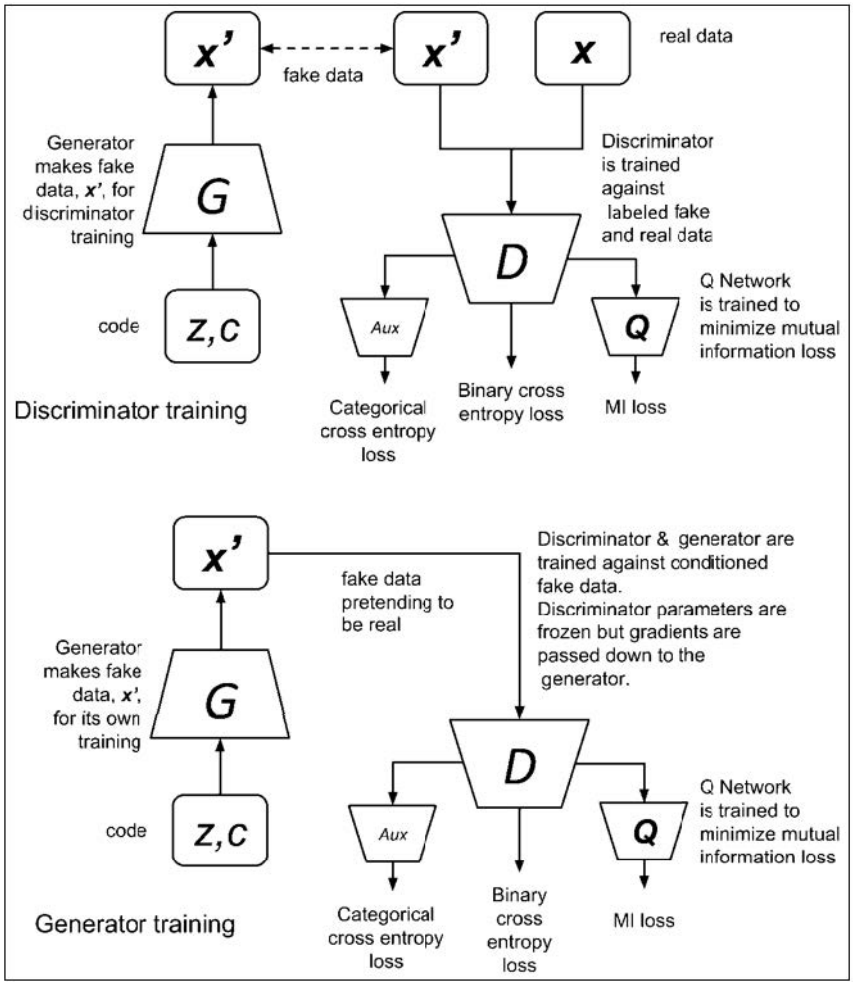

图 10.2: nfoGAN 中判别器和发生器训练的网络图

InfoGAN 的损失函数与 GAN 不同，它有一个附加项 $-\lambda I(c; g(z, c))$，其中 □ 是一个小的正常数。最小化 InfoGAN 的损失函数转化为最小化原始 GAN 的损失和最大化相互信息 $I(c; g(z, c))$。

如果应用于 MNIST 数据集，InfoGAN 可以学习离散和连续的代码，以修改发生器的输出属性。例如，像 CGAN 和 ACGAN 一样，离散码的形式是"10-dim" "one-hot" 标签，将被用来指定要生成的数字。但是，我们可以增加两个连续代码，一个用于控制书写方式的角度，另一个用于调整笔画宽度。图10.3显示了 InfoGAN 中的 MNIST 数字的代码。我们保留了维度较小的纠缠码来表示所有其他属性。

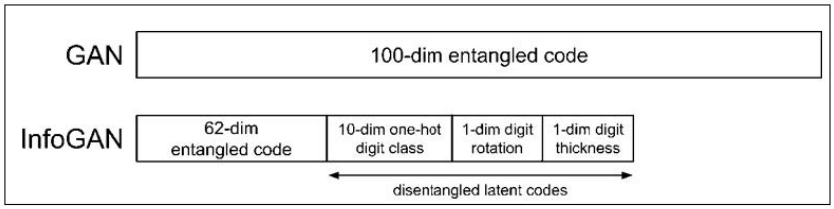

图 10.3: 在 MNIST 数据集的背景下，GAN 和 InfoGAN 的代码都是这样的

在讨论了 InfoGAN 背后的一些概念之后，让我们来看看 InfoGAN 在 tf.keras 中的实现。

在 Keras 中实现 InfoGAN

为了在 MNIST 数据集上实现 InfoGAN，需要对 ACGAN 的基本代码做一些修改。正如程序 6.1.1 所强调的，生成器将纠缠的（z 噪声代码）和不纠缠的代码（一热标签和连续代码）连接起来，作为输入。

Listing 10.2: 生成器将纠缠的和不纠缠的代码（一热标签和连续代码）连接起来，作为输入

```
inputs = [inputs, labels] + codes
```

生成器和判别器的构建函数也在 lib 文件夹下的 gan.py 中实现。

Listing 10.3: Evaluate how the model does on the test set

```
Listing 6.1.1: infogan-mnist-6.1.1.py
Highlighted are the lines that are specific to InfoGAN:
   def generator(inputs,
                 image_size,
                 activation='sigmoid',
                 labels=None,
                 codes=None):
       """Build a Generator Model
       Stack of BN-ReLU-Conv2DTranpose to generate fake images.
       Output activation is sigmoid instead of tanh in [1].
       Sigmoid converges easily.
       Arguments:
           inputs (Layer): Input layer of the generator (the z-vector)
           image_size (int): Target size of one side
               (assuming square image)
           activation (string): Name of output activation layer
           labels (tensor): Input labels
           codes (list): 2-dim disentangled codes for InfoGAN
       Returns:
           Model: Generator Model
       """
       image_resize = image_size // 4
       # network parameters
       kernel_size = 5
layer_filters = [128, 64, 32, 1]
if labels is not None:
    if codes is None:
        # ACGAN labels
        # concatenate z noise vector and one-hot labels
        inputs = [inputs, labels]
```

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

清单 6.1.2 显示了具有原始默认 GAN 输出的判别器和 Q 网络。突出显示了与离散码（用于 one-hot 标签）softmax 预测相对应的三个辅助输出和给定输入 MNIST 数字图像的连续码概率。

Listing 6.1.2: infogan-mnist-6.1.1.py Highlighted are the lines that are specific to InfoGAN:

Listing 10.4: Evaluate how the model does on the test set

```
def discriminator(inputs,
                  activation='sigmoid',
                  num_labels=None,
                  num_codes=None):
    """Build a Discriminator Model
    Stack of LeakyReLU-Conv2D to discriminate real from fake
    The network does not converge with BN so it is not used here
    unlike in [1]
    Arguments:
        inputs (Layer): Input layer of the discriminator (the image)
        activation (string): Name of output activation layer
        num_labels (int): Dimension of one-hot labels for ACGAN &
  InfoGAN
        num_codes (int): num_codes-dim Q network as output
                  if StackedGAN or 2 Q networks if InfoGAN
    Returns:
        Model: Discriminator Model
    """
```

```
        kernel_size = 5
        layer_filters = [32, 64, 128, 256]
        x = inputs
        for filters in layer_filters:
            # first 3 convolution layers use strides = 2
            # last one uses strides = 1
            if filters == layer_filters[-1]:
                strides = 1
            else:
                strides = 2
            x = LeakyReLU(alpha=0.2)(x)
            x = Conv2D(filters=filters,
                       kernel_size=kernel_size,
                        strides=strides,
                padding='same')(x)
x = Flatten()(x)
# default output is probability that the image is real
outputs = Dense(1)(x)
if activation is not None:
    print(activation)
    outputs = Activation(activation)(outputs)
if num_labels:
# ACGAN and InfoGAN have 2nd output
# 2nd output is 10-dim one-hot vector of label
layer = Dense(layer_filters[-2])(x)
labels = Dense(num_labels)(layer)
labels = Activation('softmax', name='label')(labels) if num_codes is None:
        outputs = [outputs, labels]
    else:
        # InfoGAN have 3rd and 4th outputs
        # 3rd output is 1-dim continous Q of 1st c given x
        code1 = Dense(1)(layer)
        code1 = Activation('sigmoid', name='code1')(code1)
        # 4th output is 1-dim continuous Q of 2nd c given x
        code2 = Dense(1)(layer)
        code2 = Activation('sigmoid', name='code2')(code2)
        outputs = [outputs, labels, code1, code2]
elif num_codes is not None:
    # StackedGAN Q0 output
    # z0_recon is reconstruction of z0 normal distribution
    z0_recon =  Dense(num_codes)(x)
    z0_recon = Activation('tanh', name='z0')(z0_recon)
    outputs = [outputs, z0_recon]
return Model(inputs, outputs, name='discriminator')
```

Figure 6.1.4 shows the InfoGAN model in tf.keras:

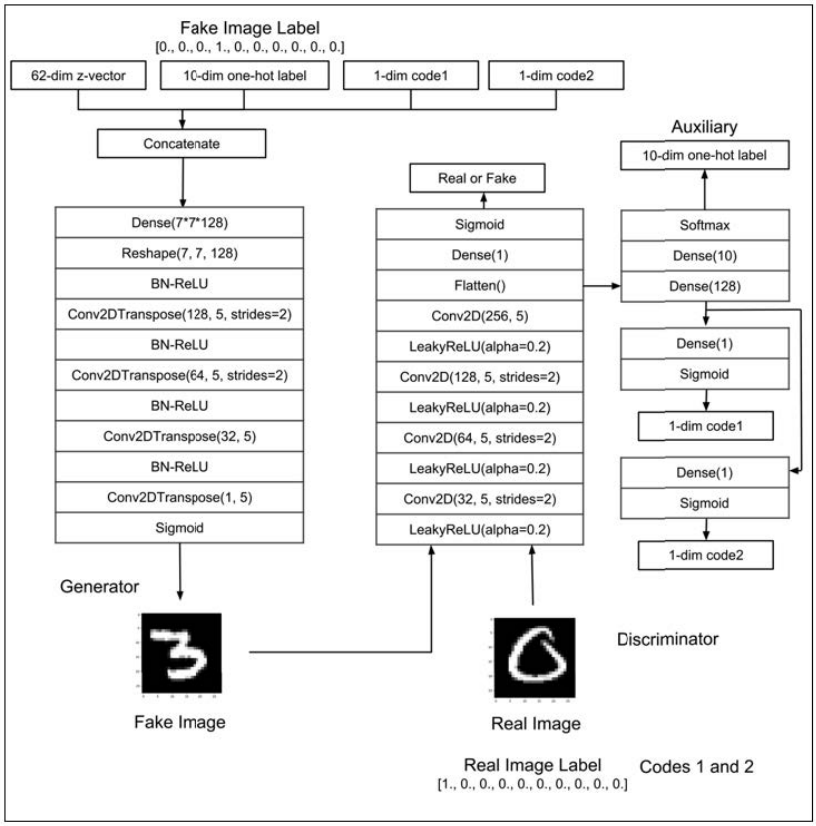

图 **10.4:** The InfoGAN Keras model

建立判别器和对抗性模型也需要一些变化。这些变化是关于所使用的损失函数的。原来的鉴别器损失函数 (binary_crossentropy)、二进制交叉熵 (categorical_crossentropy)、离散代码的分类交叉熵和每个连续代码的 mi_loss 函数组成了总体损失函数。每个损失函数的权重为 1.0，但 mi_loss 函数除外，其权重为 0.5，对应于连续代码的 $\lambda = 0.5$。

清单 6.1.3 强调了所做的改变。然而，我们应该注意到，通过使用构建器函数，判别器被实例化为:

Listing 10.5: Evaluate how the model does on the test set

```
        # call discriminator builder with 4 outputs:
        # source, label, and 2 codes
        discriminator = gan.discriminator(inputs,
                                          num_labels=num_labels,
                                          num_codes=2)
```

The generator is created by:

Listing 10.6: Evaluate how the model does on the test set

```
        # call generator with inputs,
        # labels and codes as total inputs to generator
        generator = gan.generator(inputs,
                                  image_size,
```

```
                                    labels=labels,
                                    codes=[code1, code2])
```

清单 6.1.3: infogan-mnist-6.1.1.py 互惠信息损失函数以及构建和训练 InfoGAN 判别器和对抗性网络代码:

Listing 10.7: Evaluate how the model does on the test set

```
   def mi_loss(c, q_of_c_given_x):
       """ Mutual information, Equation 5 in [2],
           assuming H(c) is constant
       """
       # mi_loss = -c * log(Q(c|x))
       return K.mean(-K.sum(K.log(q_of_c_given_x + K.epsilon()) * c,
axis=1))
   def build_and_train_models(latent_size=100):
       """Load the dataset, build InfoGAN discriminator,
       generator, and adversarial models.
       Call the InfoGAN train routine.
       """
       # load MNIST dataset
       (x_train, y_train), (_, _) = mnist.load_data()
       # reshape data for CNN as (28, 28, 1) and normalize
       image_size = x_train.shape[1]
       x_train = np.reshape(x_train, [-1, image_size, image_size, 1])
       x_train = x_train.astype('float32') / 255
       # train labels
       num_labels = len(np.unique(y_train))
       y_train = to_categorical(y_train)
       model_name = "infogan_mnist"
       # network parameters
       batch_size = 64
       train_steps = 40000
lr = 2e-4
decay = 6e-8
input_shape = (image_size, image_size, 1) label_shape = (num_labels, )
code_shape = (1, )
# build discriminator model
inputs = Input(shape=input_shape, name='discriminator_input') # call discriminator builder with 4
    outputs:
# source, label, and 2 codes
discriminator = gan.discriminator(inputs,
                                      num_labels=num_labels,
                                      num_codes=2)
       # [1] uses Adam, but discriminator converges easily with RMSprop
       optimizer = RMSprop(lr=lr, decay=decay)
       # loss functions: 1) probability image is real
       # (binary crossentropy)
       # 2) categorical cross entropy image label,
       # 3) and 4) mutual information loss
       loss = ['binary_crossentropy',
               'categorical_crossentropy',
               mi_loss,
               mi_loss]
       # lamda or mi_loss weight is 0.5
       loss_weights = [1.0, 1.0, 0.5, 0.5]
       discriminator.compile(loss=loss,
                             loss_weights=loss_weights,
```

```
                              optimizer=optimizer,
                              metrics=['accuracy'])
        discriminator.summary()
        # build generator model
        input_shape = (latent_size, )
        inputs = Input(shape=input_shape, name='z_input')
        labels = Input(shape=label_shape, name='labels')
        code1 = Input(shape=code_shape, name="code1")
        code2 = Input(shape=code_shape, name="code2")
        # call generator with inputs,
        # labels and codes as total inputs to generator
        generator = gan.generator(inputs,
                                  image_size,
                                  labels=labels,
                                  codes=[code1, code2])
        generator.summary()
# build adversarial model = generator + discriminator optimizer = RMSprop(lr=lr*0.5, decay=decay
    *0.5) discriminator.trainable = False
# total inputs = noise code, labels, and codes
        inputs = [inputs, labels, code1, code2]
        adversarial = Model(inputs,
                            discriminator(generator(inputs)),
                            name=model_name)
        # same loss as discriminator
        adversarial.compile(loss=loss,
                            loss_weights=loss_weights,
                            optimizer=optimizer,
                            metrics=['accuracy'])
        adversarial.summary()
# train discriminator and adversarial networks models = (generator, discriminator, adversarial)
    data = (x_train, y_train)
params = (batch_size,
                  latent_size,
                  train_steps,
                  num_labels,
                  model_name)
        train(models, data, params)
```

就训练而言，我们可以看到 InfoGAN 与 ACGAN 相似，只是我们需要为连续代码提供 c。c 从正态分布中抽取，标准差为 0.5，平均值为 0.0。我们将对假数据使用随机抽样的标签，对真实数据使用数据集类标签来表示离散的潜伏代码。清单 6.1.4 强调了对训练函数所作的改变。与之前所有的 GAN 类似，判别器和生成器（通过对抗性训练）被交替训练。在对抗性训练期间，鉴别器的权重被冻结。

通过使用 gan.py plot_images() 函数，每 500 个间隔步骤保存生成器输出图像样本。

Listing 10.8: Evaluate how the model does on the test set

```
Listing 6.1.4: infogan-mnist-6.1.1.py def train(models, data, params):
        """Train the Discriminator and Adversarial networks
        Alternately train discriminator and adversarial networks by batch.
        Discriminator is trained first with real and fake images,
        corresponding one-hot labels and continuous codes.
        Adversarial is trained next with fake images pretending
        to be real, corresponding one-hot labels and continous codes.
        Generate sample images per save_interval.
        # Arguments
```

```
            models (Models): Generator, Discriminator, Adversarial models
            data (tuple): x_train, y_train data
            params (tuple): Network parameters
        """
        # the GAN models
        generator, discriminator, adversarial = models
        # images and their one-hot labels
        x_train, y_train = data
        # network parameters
        batch_size, latent_size, train_steps, num_labels, model_name = \
                params
        # the generator image is saved every 500 steps
        save_interval = 500
        # noise vector to see how the generator output
        # evolves during training
        noise_input = np.random.uniform(-1.0,
                                        1.0,
                                        size=[16, latent_size])
        # random class labels and codes
noise_label = np.eye(num_labels)[np.arange(0, 16) % num_labels] noise_code1 = np.random.normal(
    scale=0.5, size=[16, 1]) noise_code2 = np.random.normal(scale=0.5, size=[16, 1])
# number of elements in train dataset
        train_size = x_train.shape[0]
        print(model_name,
              "Labels for generated images: ",
              np.argmax(noise_label, axis=1))
        for i in range(train_steps):
            # train the discriminator for 1 batch
            # 1 batch of real (label=1.0) and fake images (label=0.0)
# randomly pick real images and
# corresponding labels from dataset
rand_indexes = np.random.randint(0,
train_size,
                                 size=batch_size)
real_images = x_train[rand_indexes]
real_labels = y_train[rand_indexes]
# random codes for real images
real_code1 = np.random.normal(scale=0.5,
                              size=[batch_size, 1])
real_code2 = np.random.normal(scale=0.5,
                              size=[batch_size, 1])
# generate fake images, labels and codes
noise = np.random.uniform(-1.0,
1.0,
                          size=[batch_size, latent_size])
fake_labels = np.eye(num_labels)[np.random.choice(num_labels,
                                                  batch_size)]
fake_code1 = np.random.normal(scale=0.5,
                              size=[batch_size, 1])
fake_code2 = np.random.normal(scale=0.5,
                              size=[batch_size, 1])
inputs = [noise, fake_labels, fake_code1, fake_code2]
fake_images = generator.predict(inputs)
# real + fake images = 1 batch of train data
x = np.concatenate((real_images, fake_images)) labels = np.concatenate((real_labels, fake_labels))
    codes1 = np.concatenate((real_code1, fake_code1)) codes2 = np.concatenate((real_code2,
    fake_code2))
```

```
# label real and fake images
# real images label is 1.0
y = np.ones([2 * batch_size, 1])
# fake images label is 0.0
y[batch_size:, :] = 0
# train discriminator network,
# log the loss and label accuracy
outputs = [y, labels, codes1, codes2]
# metrics = ['loss', 'activation_1_loss', 'label_loss', # 'code1_loss', 'code2_loss', '
    activation_1_acc',
# 'label_acc', 'code1_acc', 'code2_acc']
# from discriminator.metrics_names
metrics = discriminator.train_on_batch(x, outputs)
fmt = "%d: [discriminator loss: %f, label_acc: %f]"
log = fmt % (i, metrics[0], metrics[6])
# train the adversarial network for 1 batch
# 1 batch of fake images with label=1.0 and
# corresponding one-hot label or class + random codes # since the discriminator weights are frozen
# in adversarial network only the generator is trained # generate fake images, labels and codes
noise = np.random.uniform(-1.0,
1.0,
                          size=[batch_size, latent_size])
    fake_labels = np.eye(num_labels)[np.random.choice(num_labels,
                                                      batch_size)]
    fake_code1 = np.random.normal(scale=0.5,
                                  size=[batch_size, 1])
    fake_code2 = np.random.normal(scale=0.5,
                                  size=[batch_size, 1])
    # label fake images as real
    y = np.ones([batch_size, 1])
# train the adversarial network
# note that unlike in discriminator training,
# we do not save the fake images in a variable
# the fake images go to the discriminator
# input of the adversarial for classification
# log the loss and label accuracy
inputs = [noise, fake_labels, fake_code1, fake_code2] outputs = [y, fake_labels, fake_code1,
    fake_code2] metrics = adversarial.train_on_batch(inputs, outputs) fmt = "%s [adversarial loss:
     %f, label_acc: %f]"
log = fmt % (log, metrics[0], metrics[6])
    print(log)
    if (i + 1) % save_interval == 0:
        # plot generator images on a periodic basis
        gan.plot_images(generator,
                        noise_input=noise_input,
noise_label=noise_label, noise_codes=[noise_code1, noise_code2], show=False,
step=(i + 1),
model_name=model_name)
# save the model after training the generator
# the trained generator can be reloaded for
# future MNIST digit generation
generator.save(model_name + ".h5")
```

鉴于 InfoGAN 的 tf.keras 实现，下一节介绍了生成器 MNIST 的输出，并对属性进行了拆分。

InfoGAN 的生成器输出

与之前所有呈现给我们的 GAN 类似，我们对我们的 InfoGAN 进行了 40000 步的训练。训练完成后，我们就可以运行 InfoGAN 生成器，使用保存在 infogan_mnist.h5 文件中的模型来生成新的输出。进行了以下验证。

1. 通过改变 0 到 9 的离散标签，产生数字 0 到 9, 将两个连续编码都设置为零。结果显示在图10.5中。我们可以看到，InfoGAN 的离散码可以控制发生器产生的数字。

   执行方法：

   python3 infogan-mnist-6.1.1.py –generator=infogan_mnist.h5

   –digit=0 –code1=0 –code2=0

   到

   python3 infogan-mnist-6.1.1.py –generator=infogan_mnist.h5

   –digit=9 –code1=0 –code2=0

   在图10.5中，可以看到由 InfoGAN 生成的图像。

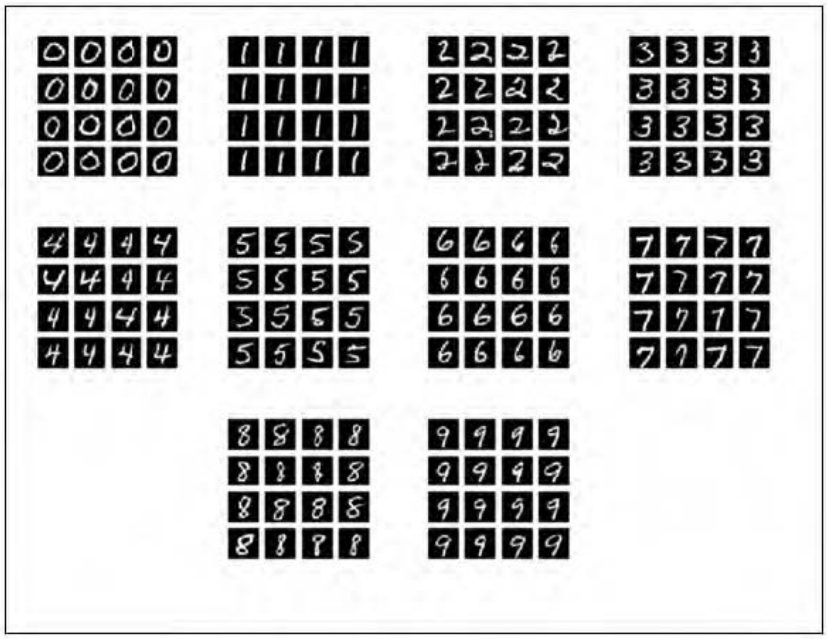

图 10.5: 由 InfoGAN 生成的图像作为离散代码从 0 到 9 变化, 将两个连续编码都被设置为零

2. 检查第一个连续码的影响，以了解哪个属性受到影响。将第一个连续码从-2.0 到 2.0 变化为数字 0 到 9。第二个连续码被设置为 0.0。图10.6显示，第一个连续码控制了数字的粗细。

   python3 infogan-mnist-6.1.1.py –generator=infogan_mnist.h5

   –digit=0 –code1=0 –code2=0 –p1

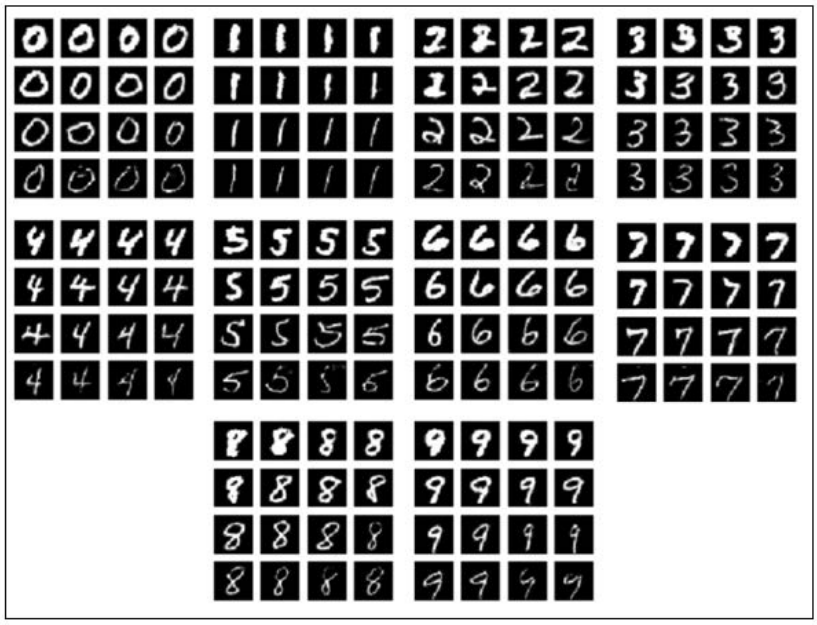

图 10.6: 由 InfoGAN 生成的图像作为第一个连续码，数字 0 到 9 从-2.0 到 2.0 变化。第二个连续码被设置为零。第一个连续码控制数字的粗细

3. 与上一步类似，而是更加关注第二个连续代码。图10.7显示，第二个连续代码控制书写方式的旋转角度（倾斜）。

   python3 infogan-mnist-6.1.1.py –generator=infogan_mnist.h5

   –digit=0 –code1=0 –code2=0 –p2

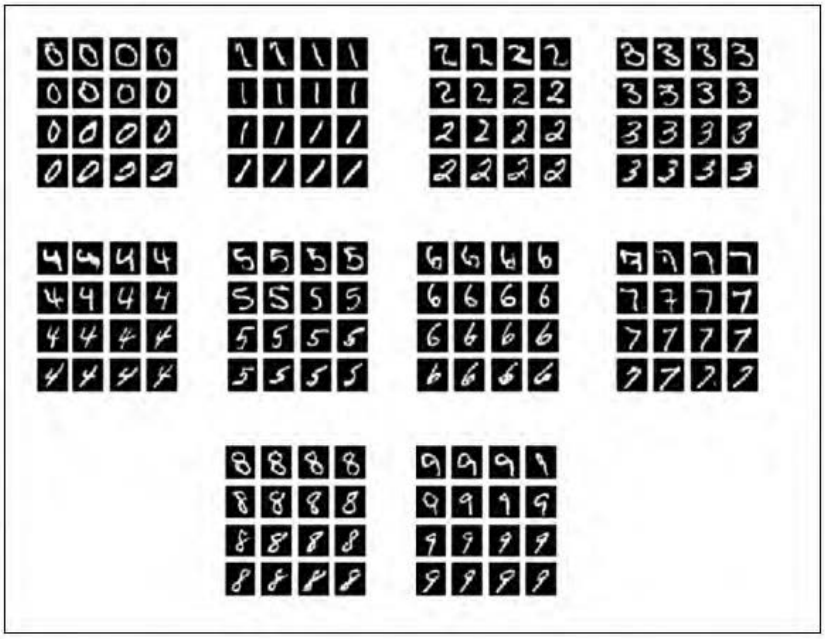

图 10.7: 图10.7：当第二个连续码从-2.0 到 2.0 变化时，数字 0 到 9 由 InfoGAN 生成的图像。第一个连续码被设置为零。第二个连续码控制书写方式的旋转角度（倾斜）。

从这些验证结果中，我们可以看到，除了能够生成 MNIST 外观的数字外，InfoGAN 还扩展了 CGAN 和 ACGAN 等条件 GAN 的能力。该网络自动学习了两个任意的代码，可以控制生成器输出的具体属性。如果我们把连续代码的数量增加到 2 个以上，看看还能控制哪些额外的属性将是很有趣的。这可以通过增加清单 6.1.1 至清单 6.1.4 中高亮行的代码列表来实现。

本节的结果表明，生成器输出的属性可以通过最大化代码和数据分布之间的相互信息来分解。在下一节中，将介绍一种不同的解缠方法。StackedGAN 的想法是在特征层面上注入代码。

10.3. 叠加 GAN

本着与 InfoGAN 相同的精神，StackedGAN 提出了一种为调节生成器输出的潜在表征进行分解的方法。然而，StackedGAN 使用了一种不同的方法来解决这个问题。StackedGAN 不是学习如何调节噪声以产生所需的输出，而是将一个 GAN 分解为一叠 GAN。每个 GAN 以通常的判别器-对抗方式独立训练，有自己的潜伏代码。

图10.8向我们展示了 StackedGAN 在假设的名人脸部生成的背景下是如何工作的，假设编码器网络已经被训练为名人脸部分类。

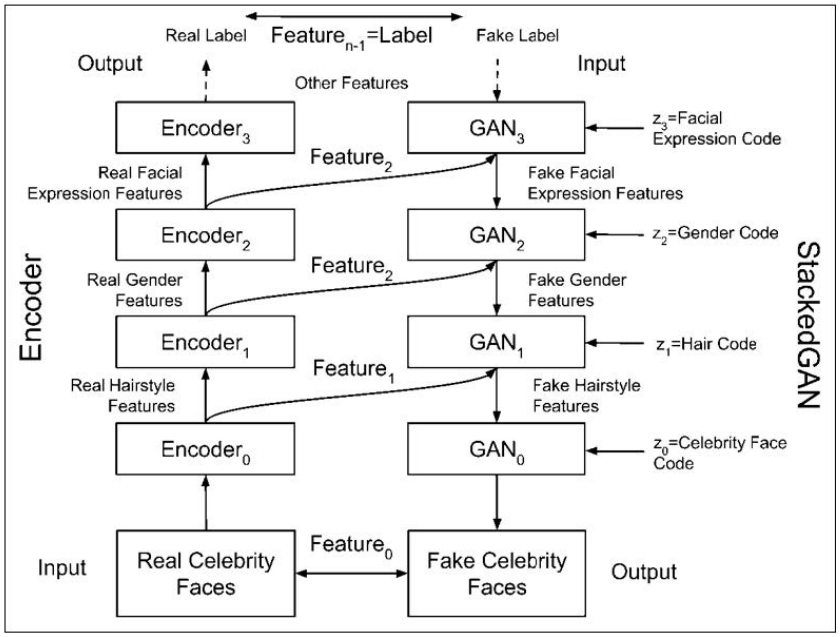

图 10.8: 在名人面孔生成的背景下，StackedGAN 的基本理念。假设有一个假设的深度编码器网络可以对名人脸部进行分类，StackedGAN 只是简单地将编码器的过程颠倒过来

编码器网络是由一堆简单的编码器组成的，其中 $i = 0 \ldots n-1$ 对应 n 个特征。每个编码器提取某些面部特征。例如，编码器 0 可能是发型特征的编码器，特征 1。所有的简单编码器都有助于使整个编码器进行正确的预测。

StackedGAN 背后的想法是，如果我们想建立一个能生成假的名人脸的 GAN，我们应该简单地反转编码器。StackedGAN 由一堆较简单的 GAN 组成，GANi，其中 $i = 0 \ldots n-1$，对应 n 个特征。每个 GAN_i 学习反转其相应编码器 Encoderi 的过程。例如，GAN_0 从假的发型特征生成假的名人面孔，这是 Encoder0 过程的反转。

每个 GAN_i 都使用一个潜伏代码，即 z_i，来制约其生成器的输出。例如，潜伏代码，z_0，可以将发型从卷发改为波浪形。$GANs$ 的堆栈也可以作为一个整体来合成假的名人脸，完成整个编码器的逆过程。每个 GANi 的潜伏代码，Z_i，可以用来改变假名人脸的特定属性。

有了 StackedGAN 工作原理的关键想法，让我们进入下一节，看看它是如何在 tf.keras 中实现的。

10.4. Keras 中的 StackedGAN 的实现

图10.9中可以看到一个堆叠式 GAN 的详细网络模型。为了简洁起见，每个堆栈只显示了两个编码器 encoder-GANs。该图最初可能看起来很复杂，但它只是一个编码器 encoder-GAN 的重复，这意味着如果我们理解了如何训练一个编码器 encoder-GANs，其余的也会利用同样的概念。

在本节中，我们假设 StackedGAN 是为 MNIST 数字生成而设计的。

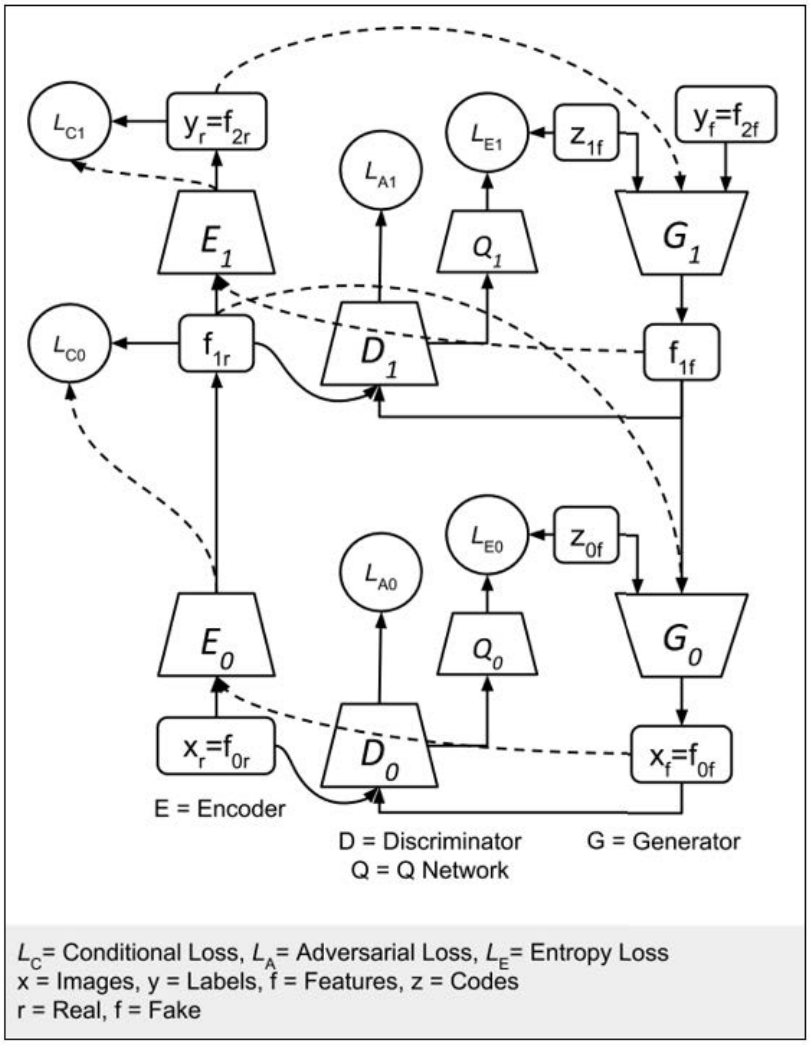

图 10.9: StackedGAN 包括一个编码器和一个 GAN 的堆栈。编码器经过预训练以进行分类。$Generator'_1$，G'_1，学习以假标签 f_{1f} 和潜伏代码 z_{1f} 为条件合成 f_{f1} 特征。$Generator_{0'}$，$G_{0'}$，使用假特征，f_{1f} 和潜伏代码，z_{0f} 来生成假图像。

StackedGAN 以一个编码器开始。它可以是一个经过训练的分类器，预测正确的标签。中间的特征向量 f_{1r} 被提供给 GAN 训练。对于 MNIST，可以使用一个基于 CNN 的分类器。

图10.10显示了编码器及其在 tf.keras 中的网络模型实现。

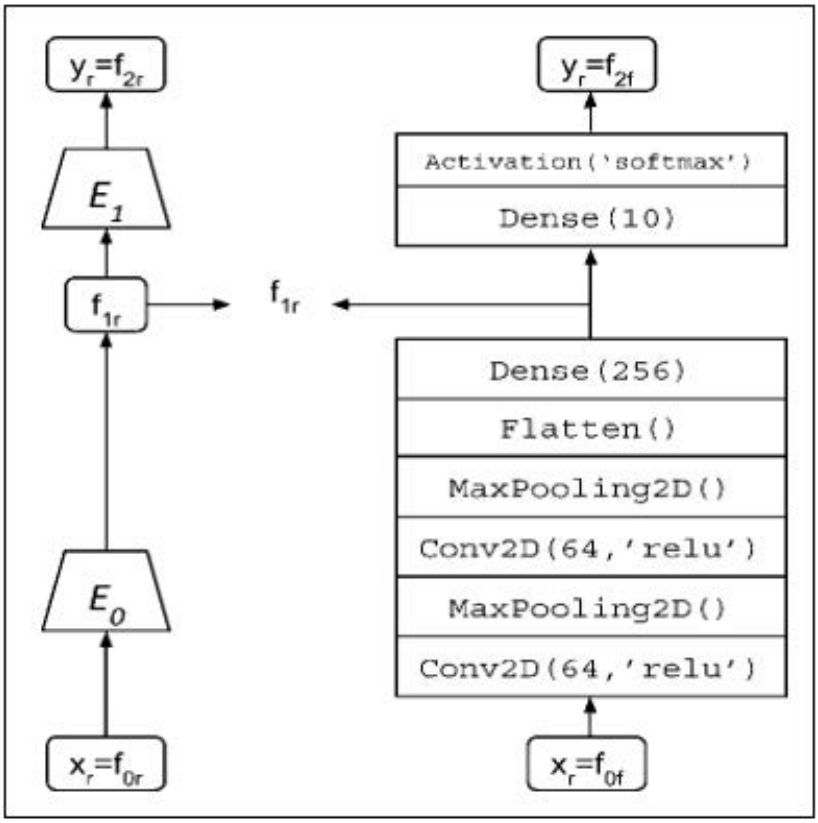

图 10.10: StackedGAN 中的编码器是一个简单的基于 CNN 的分类器

清单 6.2.1 显示了上图的 tf.keras 代码。它与基于 CNN 的分类器类似，只是我们用 Dense 层来提取 256 分的特征。有两个输出模型，$Encoder_0$ 和 $Encoder_1$。两者都将被用来训练 StackedGAN。

Listing 10.9: Listing 6.2.1: stackedgan-mnist-6.2.1.py

```
Listing 6.2.1: stackedgan-mnist-6.2.1.py
def build_encoder(inputs, num_labels=10, feature1_dim=256):
       """ Build the Classifier (Encoder) Model sub networks
       Two sub networks:
       1) Encoder0: Image to feature1 (intermediate latent feature)
       2) Encoder1: feature1 to labels
       # Arguments
           inputs (Layers): x - images, feature1 -
               feature1 layer output
           num_labels (int): number of class labels
           feature1_dim (int): feature1 dimensionality
       # Returns
           enc0, enc1 (Models): Description below
       """
       kernel_size = 3
       filters = 64
       x, feature1 = inputs
       # Encoder0 or enc0
       y = Conv2D(filters=filters,
```

```
                kernel_size=kernel_size,
                padding='same',
                activation='relu')(x)
    y = MaxPooling2D()(y)
    y = Conv2D(filters=filters,
                kernel_size=kernel_size,
                padding='same',
                activation='relu')(y)
    y = MaxPooling2D()(y)
    y = Flatten()(y)
    feature1_output = Dense(feature1_dim, activation='relu')(y)
    # Encoder0 or enc0: image (x or feature0) to feature1
    enc0 = Model(inputs=x, outputs=feature1_output, name="encoder0")
    # Encoder1 or enc1
    y = Dense(num_labels)(feature1)
    labels = Activation('softmax')(y)
    # Encoder1 or enc1: feature1 to class labels (feature2)
    enc1 = Model(inputs=feature1, outputs=labels, name="encoder1")
    # return both enc0 and enc1
    return enc0, enc1
```

$Encoder_0$ 的输出 $f_(1r)$，是我们希望 $Generator_1$ 学会合成的 256 维特征向量。它可以作为 $Encoder_0$ 的一个辅助输出 E_0。整个编码器被训练来对 MNIST 数字进行分类 x_r。正确的标签 y_r，是由 $Encoder_1$，E_1 预测的。在这个过程中，中间的特征集 f_{1r} 被学习并提供给 $Generator_0$ 训练。当 GAN 针对这个编码器进行训练时，下标 r 被用来强调和区分真实数据和虚假数据。

考虑到编码器输入（xr）的中间特征（f1r）和标签（yr），每个 GAN 都以通常的判别器对抗方式进行训练。损失函数由表 6.2.1 中的公式 6.2.1 至公式 6.2.5 给出。方程 6.2.1 和方程 6.2.2 是通用 GAN 的通常损失函数。StackedGAN 有两个额外的损失函数，即条件型和熵型。

公式 6.2.3 中的条件损失函数 $L_i^{(G)cond}$ 确保发生器在从输入噪声码 zi 合成输出 fi 时不会忽略输入 f_{i+1}。编码器 $Encoder_i$ 必须能够通过反转发生器 $Generator_i$ 的过程来恢复发生器的输入。发电机输入和使用编码器恢复的输入之间的差异用 $L2$ 或欧氏距离（平均平方误差（MSE））来衡量。

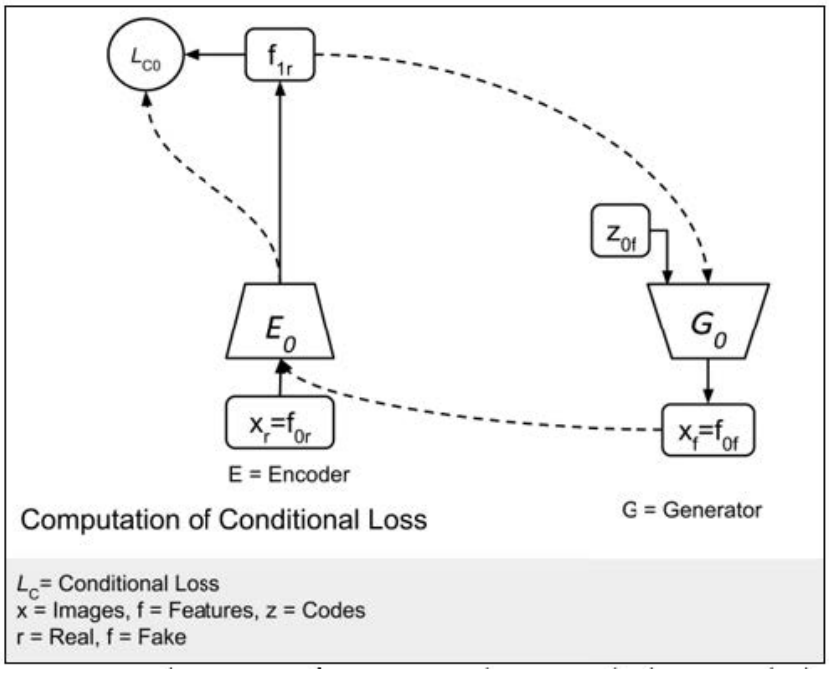

图 10.11: 图10.10的一个简单版本只显示了参与 $L_0^{(G)cond}$ 计算的网络元素

然而，条件损失函数引入了一个新的问题。生成器忽略了输入的噪声代码，z_i，而仅仅依赖于 f_{i+1}。熵损失函数 $L_i^{(G)ent}$ 方程 6.2.4 中，确保发生器不会忽略噪声代码 z_i。Q 网络从发生器的输出中恢复了噪声代码。恢复的噪声和输入的噪声之间的差异也由 $L2$ 或欧氏距离（MSE）来衡量。

图10.12显示了参与 $L_0^{(G)ent}$ 计算的网络元素。

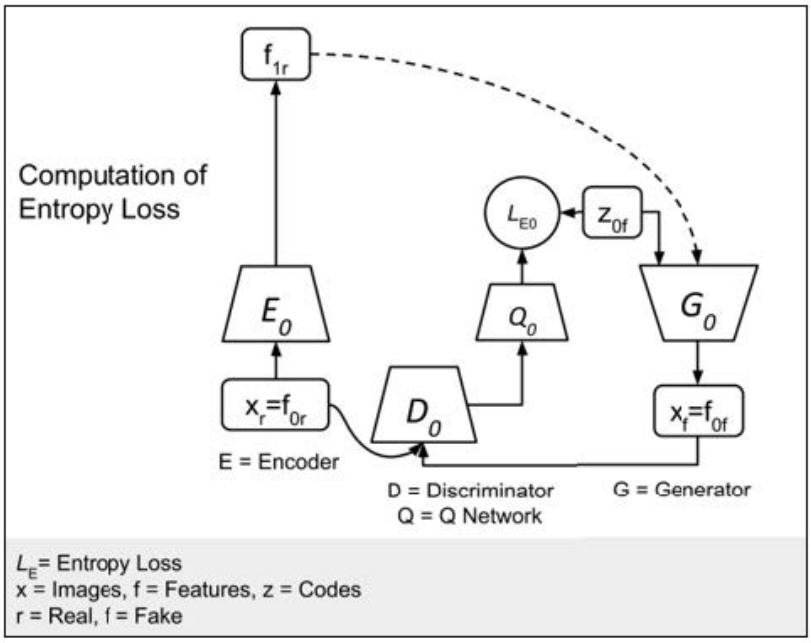

图 10.12: 图10.10的简单版本只向我们展示了参与 $L_0^{(G)ent}$ 计算的网络元素。

最后一个损失函数类似于通常的 GAN 损失。它包括判别器损失，$L_i^{(D)}$ ，和生成器（通过对抗性）损失，$L_i^{(D)adv}$ 。图10.13显示了 GAN 损失中涉及的元素。

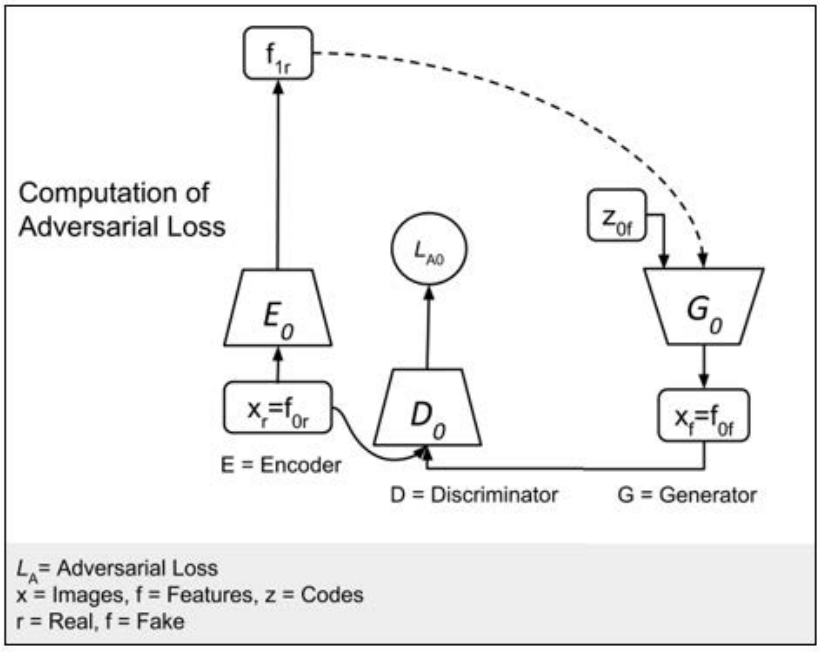

图 10.13: 图10.10的一个简单版本，只显示参与计算的网络元素

在公式 6.2.5 中，三个生成器损失函数的加权和就是最终的生成器损失函数。在我们将要介绍的 Keras 代码中，所有的权重都被设置为 1.0，除了熵损失，它被设置为 10.0。在公式 6.2.1 到公式 6.2.5 中，i 指的是编码器和 GAN 组的 ID 或级别。在原论文中，网络首先是独立训练，然后是联合训练。在独立训练期间，首先训练编码器。在联合训练期间，同时使用真实和虚假数据。

在 tf.keras 中实现 StackedGAN 生成器和判别器需要做一些改变，以提供辅助点来访问中间特征。图 6.2.7 显示了发生器 tf.keras 模型。

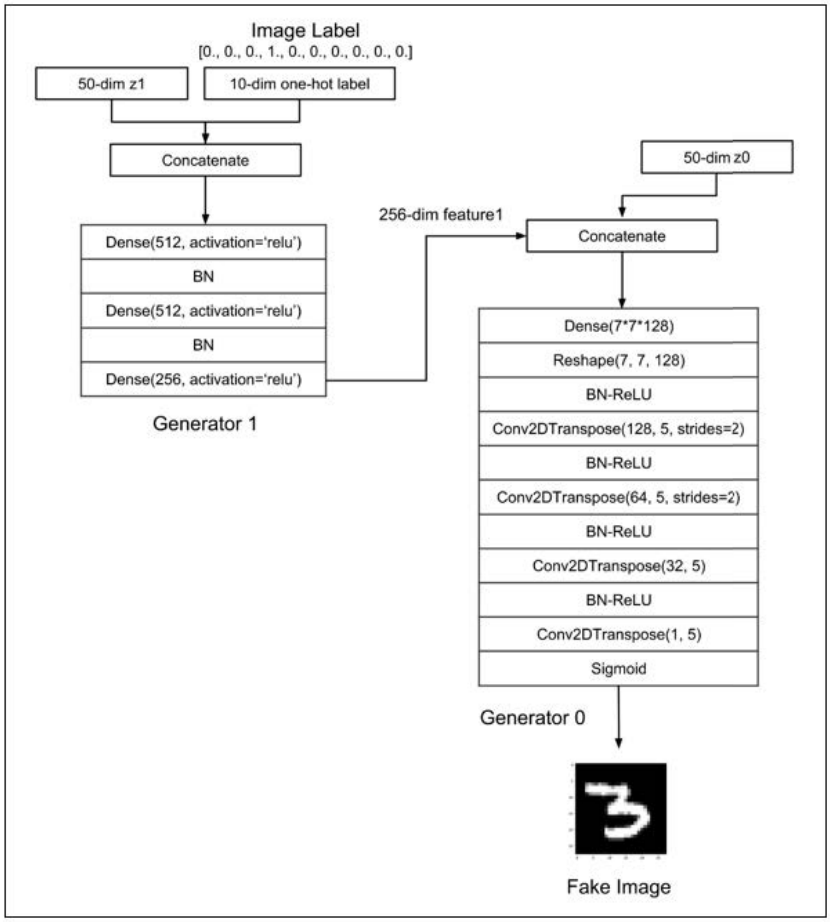

图 10.14: Keras 中的 StackedGAN 生成器模型

Listing 6.2.2 illustrates the function that builds two generators (gen0 and gen1) corresponding to Generator0 and Generator1. The gen1 generator is made of three Dense layers with labels and the noise code z1f as inputs. The third layer generates the fake f1f feature. The gen0 generator is similar to other GAN generators that we've presented and can be instantiated using the generator builder in gan.py:

清单 6.2.2 说明了建立两个生成器（gen_0 和 gen_1）的函数，对应于 $Generator_0$ 和 $Generator_1$。gen_1 生成器由三个 Dense 层组成，标签和噪声代码 z_{1f} 作为输入。第三层生成假的 f_{1f} 特征。$gen0$ 生成器与我们介绍过的其他 GAN 生成器相似，可以使用 gan.py 中的生成器生成器来实例化。

Listing 10.10: gan.py

```
    # gen0: feature1 + z0 to feature0 (image)
    gen0 = gan.generator(feature1, image_size, codes=z0)
```

$gen0$ 的输入是 f_1 特征和噪声代码 z_0。输出是生成的假图像，x_f。

Listing 10.11: Listing 6.2.2: stackedgan-mnist-6.2.1.py

```
Listing 6.2.2: stackedgan-mnist-6.2.1.py
def build_generator(latent_codes, image_size, feature1_dim=256):
        """Build Generator Model sub networks
        Two sub networks: 1) Class and noise to feature1
            (intermediate feature)
            2) feature1 to image
        # Arguments
            latent_codes (Layers): dicrete code (labels),
                noise and feature1 features
            image_size (int): Target size of one side
                (assuming square image)
            feature1_dim (int): feature1 dimensionality
        # Returns
            gen0, gen1 (Models): Description below
"""
        # Latent codes and network parameters
        labels, z0, z1, feature1 = latent_codes
        # image_resize = image_size // 4
        # kernel_size = 5
        # layer_filters = [128, 64, 32, 1]
        # gen1 inputs
        inputs = [labels, z1]      # 10 + 50 = 62-dim
        x = concatenate(inputs, axis=1)
        x = Dense(512, activation='relu')(x)
        x = BatchNormalization()(x)
        x = Dense(512, activation='relu')(x)
        x = BatchNormalization()(x)
        fake_feature1 = Dense(feature1_dim, activation='relu')(x)
        # gen1: classes and noise (feature2 + z1) to feature1
        gen1 = Model(inputs, fake_feature1, name='gen1')
        # gen0: feature1 + z0 to feature0 (image)
        gen0 = gan.generator(feature1, image_size, codes=z0)
return gen0, gen1
```

图10.15显示了判别器 tf.keras 模型。

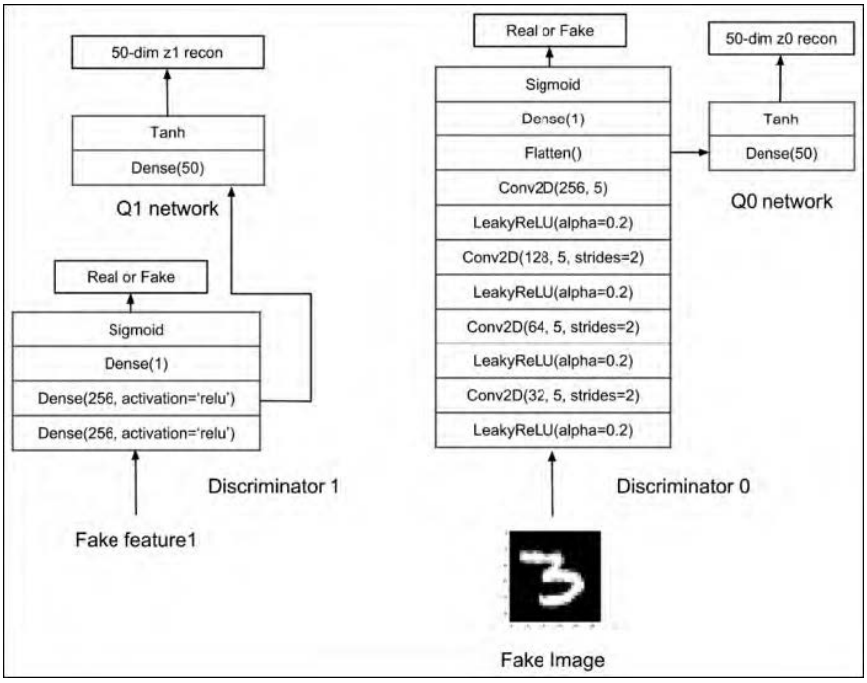

图 10.15: Keras 中的 StackedGAN 判别器模型

我们提供了建立 $Discriminator_0$ 和 $Discriminator_1$（$dis0$ 和 $dis1$）的函数。$dis0$ 判别器类似于 GAN 判别器，除了特征向量输入和恢复 z_0 的辅助网络 Q_0。gan.py 中的 builder 函数被用来创建 $dis0$。

Listing 10.12: Evaluate how the model does on the test set

```
dis0 = gan.discriminator(inputs, num_codes=z_dim)
```

如清单10.10所示，dis1 鉴别器是由三层 MLP 组成。最后一层是对真假 f_1 的判别。Q_1 网络共享 dis1 的前两层。它的第三层恢复了 z_1。

Listing 10.13: Listing 6.2.3: stackedgan-mnist-6.2.1.py

```
Listing 6.2.3: stackedgan-mnist-6.2.1.py
def build_discriminator(inputs, z_dim=50):
       """Build Discriminator 1 Model
       Classifies feature1 (features) as real/fake image and recovers
       the input noise or latent code (by minimizing entropy loss)
       # Arguments
           inputs (Layer): feature1
           z_dim (int): noise dimensionality
       # Returns
           dis1 (Model): feature1 as real/fake and recovered latent code
"""
       # input is 256-dim feature1
       x = Dense(256, activation='relu')(inputs)
       x = Dense(256, activation='relu')(x)
       # first output is probability that feature1 is real
       f1_source = Dense(1)(x)
       f1_source = Activation('sigmoid',
```

```
                              name='feature1_source')(f1_source)
        # z1 reonstruction (Q1 network)
        z1_recon = Dense(z_dim)(x)
        z1_recon = Activation('tanh', name='z1')(z1_recon)
        discriminator_outputs = [f1_source, z1_recon]
        dis1 = Model(inputs, discriminator_outputs, name='dis1')
        return dis1
```

在所有构建函数可用的情况下，StackedGAN 在清单 6.2.4 中被组装起来。在训练 StackedGAN 之前，对编码器进行预训练。请注意，我们已经在对抗性模型训练中加入了三个生成器损失函数（对抗性、条件性和熵）。Q 网络与鉴别器模型共享一些共同层。因此，它的损失函数也被纳入鉴别器模型的训练中。

Listing 10.14: Listing 6.2.4: stackedgan-mnist-6.2.1.py

```
Listing 6.2.4: stackedgan-mnist-6.2.1.py
    def build_and_train_models():
        """Load the dataset, build StackedGAN discriminator,
        generator, and adversarial models.
        Call the StackedGAN train routine.
        """
        # load MNIST dataset
        (x_train, y_train), (x_test, y_test) = mnist.load_data()
        # reshape and normalize images
        image_size = x_train.shape[1]
        x_train = np.reshape(x_train, [-1, image_size, image_size, 1])
        x_train = x_train.astype('float32') / 255
        x_test = np.reshape(x_test, [-1, image_size, image_size, 1])
        x_test = x_test.astype('float32') / 255
        # number of labels
        num_labels = len(np.unique(y_train))
        # to one-hot vector
        y_train = to_categorical(y_train)
        y_test = to_categorical(y_test)
        model_name = "stackedgan_mnist"
        # network parameters
        batch_size = 64
        train_steps = 10000
        lr = 2e-4
        decay = 6e-8
        input_shape = (image_size, image_size, 1)
        label_shape = (num_labels, )
        z_dim = 50
        z_shape = (z_dim, )
        feature1_dim = 256
        feature1_shape = (feature1_dim, )
        # build discriminator 0 and Q network 0 models
        inputs = Input(shape=input_shape, name='discriminator0_input')
        dis0 = gan.discriminator(inputs, num_codes=z_dim)
        # [1] uses Adam, but discriminator converges easily with RMSprop
        optimizer = RMSprop(lr=lr, decay=decay)
        # loss fuctions: 1) probability image is real (adversarial0 loss)
        # 2) MSE z0 recon loss (Q0 network loss or entropy0 loss)
        loss = ['binary_crossentropy', 'mse']
        loss_weights = [1.0, 10.0]
        dis0.compile(loss=loss,
                     loss_weights=loss_weights,
```

```
            optimizer=optimizer,
            metrics=['accuracy'])
dis0.summary() # image discriminator, z0 estimator
# build discriminator 1 and Q network 1 models
input_shape = (feature1_dim, )
inputs = Input(shape=input_shape, name='discriminator1_input')
dis1 = build_discriminator(inputs, z_dim=z_dim )
# loss fuctions: 1) probability feature1 is real
# (adversarial1 loss)
# 2) MSE z1 recon loss (Q1 network loss or entropy1 loss)
loss = ['binary_crossentropy', 'mse']
loss_weights = [1.0, 1.0]
dis1.compile(loss=loss,
            loss_weights=loss_weights,
            optimizer=optimizer,
            metrics=['accuracy'])
dis1.summary() # feature1 discriminator, z1 estimator
# build generator models
feature1 = Input(shape=feature1_shape, name='feature1_input')
labels = Input(shape=label_shape, name='labels')
z1 = Input(shape=z_shape, name="z1_input")
z0 = Input(shape=z_shape, name="z0_input")
latent_codes = (labels, z0, z1, feature1)
gen0, gen1 = build_generator(latent_codes, image_size)
gen0.summary() # image generator
gen1.summary() # feature1 generator
# build encoder models
input_shape = (image_size, image_size, 1)
inputs = Input(shape=input_shape, name='encoder_input')
enc0, enc1 = build_encoder((inputs, feature1), num_labels)
enc0.summary() # image to feature1 encoder
enc1.summary() # feature1 to labels encoder (classifier)
encoder = Model(inputs, enc1(enc0(inputs)))
encoder.summary() # image to labels encoder (classifier)
data = (x_train, y_train), (x_test, y_test)
train_encoder(encoder, data, model_name=model_name)
# build adversarial0 model =
# generator0 + discriminator0 + encoder0
optimizer = RMSprop(lr=lr*0.5, decay=decay*0.5)
       # encoder0 weights frozen
       enc0.trainable = False
       # discriminator0 weights frozen
       dis0.trainable = False
       gen0_inputs = [feature1, z0]
       gen0_outputs = gen0(gen0_inputs)
       adv0_outputs = dis0(gen0_outputs) + [enc0(gen0_outputs)]
       # feature1 + z0 to prob feature1 is
       # real + z0 recon + feature0/image recon
       adv0 = Model(gen0_inputs, adv0_outputs, name="adv0")
       # loss functions: 1) prob feature1 is real (adversarial0 loss)
       # 2) Q network 0 loss (entropy0 loss)
       # 3) conditional0 loss
       loss = ['binary_crossentropy', 'mse', 'mse']
       loss_weights = [1.0, 10.0, 1.0]
       adv0.compile(loss=loss,
                    loss_weights=loss_weights,
                    optimizer=optimizer,
```

```
                    metrics=['accuracy'])
        adv0.summary()
        # build adversarial1 model =
        # generator1 + discriminator1 + encoder1
        # encoder1 weights frozen
        enc1.trainable = False
        # discriminator1 weights frozen
        dis1.trainable = False
        gen1_inputs = [labels, z1]
        gen1_outputs = gen1(gen1_inputs)
        adv1_outputs = dis1(gen1_outputs) + [enc1(gen1_outputs)]
        # labels + z1 to prob labels are real + z1 recon + feature1 recon
        adv1 = Model(gen1_inputs, adv1_outputs, name="adv1")
        # loss functions: 1) prob labels are real (adversarial1 loss)
        # 2) Q network 1 loss (entropy1 loss)
        # 3) conditional1 loss (classifier error)
        loss_weights = [1.0, 1.0, 1.0]
        loss = ['binary_crossentropy',
'mse',
                'categorical_crossentropy']
        adv1.compile(loss=loss,
                     loss_weights=loss_weights,
                     optimizer=optimizer,
                     metrics=['accuracy'])
        adv1.summary()
        # train discriminator and adversarial networks
        models = (enc0, enc1, gen0, gen1, dis0, dis1, adv0, adv1)
        params = (batch_size, train_steps, num_labels, z_dim, model_name)
        train(models, data, params)
```

最后，训练函数与典型的 GAN 训练有相似之处，只是我们一次只训练一个 GAN（即先训练 GAN_1，再训练 GAN_0）。代码显示在清单 6.2.5 中。值得注意的是，训练的顺序是。

1. 通过最小化判别器和熵损失来训练判别器 $Discriminator_1$ 和 Q_1 网络；
2. 通过最小化判别器和熵损失来判别器 $Discriminator_0$ 和 Q_0 网络；
3. 通过最小化对抗性、熵和条件性损失来实现对抗性网络 $Adversarial_1$；
4. 通过最小化对抗性、熵和条件性损失，对抗性网络 $Adversarial0$。

Listing 10.15: Listing 6.2.5: stackedgan-mnist-6.2.1.py

```
def train(models, data, params):
        """Train the discriminator and adversarial Networks
        Alternately train discriminator and adversarial networks by batch.
        Discriminator is trained first with real and fake images,
        corresponding one-hot labels and latent codes.
        Adversarial is trained next with fake images pretending
        to be real, corresponding one-hot labels and latent codes.
        Generate sample images per save_interval.
        # Arguments
            models (Models): Encoder, Generator, Discriminator,
                Adversarial models
            data (tuple): x_train, y_train data
            params (tuple): Network parameters
        """
        # the StackedGAN and Encoder models
```

```
        enc0, enc1, gen0, gen1, dis0, dis1, adv0, adv1 = models
        # network parameters
        batch_size, train_steps, num_labels, z_dim, model_name = params
        # train dataset
        (x_train, y_train), (_, _) = data
        # the generator image is saved every 500 steps
        save_interval = 500
        # label and noise codes for generator testing
        z0 = np.random.normal(scale=0.5, size=[16, z_dim])
        z1 = np.random.normal(scale=0.5, size=[16, z_dim])
        noise_class = np.eye(num_labels)[np.arange(0, 16) % num_labels]
        noise_params = [noise_class, z0, z1]
        # number of elements in train dataset
        train_size = x_train.shape[0]
        print(model_name,
              "Labels for generated images: ",
              np.argmax(noise_class, axis=1))
        for i in range(train_steps):
            # train the discriminator1 for 1 batch
            # 1 batch of real (label=1.0) and fake feature1 (label=0.0)
            # randomly pick real images from dataset
            rand_indexes = np.random.randint(0,
train_size,
                                             size=batch_size)
            real_images = x_train[rand_indexes]
            # real feature1 from encoder0 output
            real_feature1 = enc0.predict(real_images)
            # generate random 50-dim z1 latent code
            real_z1 = np.random.normal(scale=0.5,
size=[batch_size, z_dim])
real_labels = y_train[rand_indexes]
# generate fake feature1 using generator1 from
# real labels and 50-dim z1 latent code
fake_z1 = np.random.normal(scale=0.5,
                           size=[batch_size, z_dim])
fake_feature1 = gen1.predict([real_labels, fake_z1])
# real + fake data
feature1 = np.concatenate((real_feature1, fake_feature1))
z1 = np.concatenate((fake_z1, fake_z1))
# label 1st half as real and 2nd half as fake
y = np.ones([2 * batch_size, 1])
y[batch_size:, :] = 0
         # train discriminator1 to classify feature1 as
         # real/fake and recover
         # latent code (z1). real = from encoder1,
         # fake = from genenerator1
         # joint training using discriminator part of
         # advserial1 loss and entropy1 loss
         metrics = dis1.train_on_batch(feature1, [y, z1])
         # log the overall loss only
         log = "%d: [dis1_loss: %f]" % (i, metrics[0])
         # train the discriminator0 for 1 batch
         # 1 batch of real (label=1.0) and fake images (label=0.0)
         # generate random 50-dim z0 latent code
         fake_z0 = np.random.normal(scale=0.5, size=[batch_size, z_
dim])
         # generate fake images from real feature1 and fake z0
```

```
        fake_images = gen0.predict([real_feature1, fake_z0])
        # real + fake data
        x = np.concatenate((real_images, fake_images))
        z0 = np.concatenate((fake_z0, fake_z0))
        # train discriminator0 to classify image
        # as real/fake and recover latent code (z0)
        # joint training using discriminator part of advserial0 loss
        # and entropy0 loss
        metrics = dis0.train_on_batch(x, [y, z0])
        # log the overall loss only (use dis0.metrics_names)
        log = "%s [dis0_loss: %f]" % (log, metrics[0])
        # adversarial training
        # generate fake z1, labels
        fake_z1 = np.random.normal(scale=0.5,
                                   size=[batch_size, z_dim])
        # input to generator1 is sampling fr real labels and
        # 50-dim z1 latent code
        gen1_inputs = [real_labels, fake_z1]
        # label fake feature1 as real
        y = np.ones([batch_size, 1])
        # train generator1 (thru adversarial) by fooling i
           # the discriminator
           # and approximating encoder1 feature1 generator
           # joint training: adversarial1, entropy1, conditional1
           metrics = adv1.train_on_batch(gen1_inputs,
                                         [y, fake_z1, real_labels])
           fmt = "%s [adv1_loss: %f, enc1_acc: %f]"
           # log the overall loss and classification accuracy
           log = fmt % (log, metrics[0], metrics[6])
           # input to generator0 is real feature1 and
           # 50-dim z0 latent code
           fake_z0 = np.random.normal(scale=0.5,
                                      size=[batch_size, z_dim])
           gen0_inputs = [real_feature1, fake_z0]
           # train generator0 (thru adversarial) by fooling
           # the discriminator and approximating encoder1 imag
           # source generator joint training:
           # adversarial0, entropy0, conditional0
           metrics = adv0.train_on_batch(gen0_inputs,
[y, fake_z0, real_feature1])
    log = "%s [adv0_loss: %f]" % (log, metrics[0])
    print(log)
    if (i + 1) % save_interval == 0:
        generators = (gen0, gen1)
        plot_images(generators,
                    noise_params=noise_params,
                    show=False,
                    step=(i + 1),
                    model_name=model_name)
# save the modelis after training generator0 & 1
# the trained generator can be reloaded for
# future MNIST digit generation
gen1.save(model_name + "-gen1.h5")
gen0.save(model_name + "-gen0.h5")
```

tf.keras 中 StackedGAN 的代码实现现在已经完成。训练结束后，可以对生成器的输出进行评估，以考察合成的 MNIST 数字的某些属性是否可以用类似于我们在 InfoGAN 中的方式来控制。

10.4.1. StackedGAN 的生成器输出

在对 StackedGAN 进行 10000 步训练后，$Generator_0$ 和 $Generator_1$ 模型被保存在文件中。叠加在一起，$Generator_0$ 和 $Generator_1$ 可以合成以标签和噪声代码 z_0 和 z_1 为条件的假图像。StackedGAN 发生器可以通过以下方式进行定性验证。

1. 将离散的标签从 0 到 9 变化，这两个噪声代码，z_0 和 z_1 从平均数为 0.5 和标准差为 1.0 的正态分布中采样。结果显示在图10.16中。我们能够看到，StackedGAN 的离散码能够控制发生器产生的数字。

   python3 stackedgan-mnist-6.2.1.py

   –generator0=stackedgan_mnist-gen0.h5

   –generator1=stackedgan_mnist-gen1.h5 –digit=0

   到 python3 stackedgan-mnist-6.2.1.py

   –generator0=stackedgan_mnist-gen0.h5

   –generator1=stackedgan_mnist-gen1.h5 –digit=9

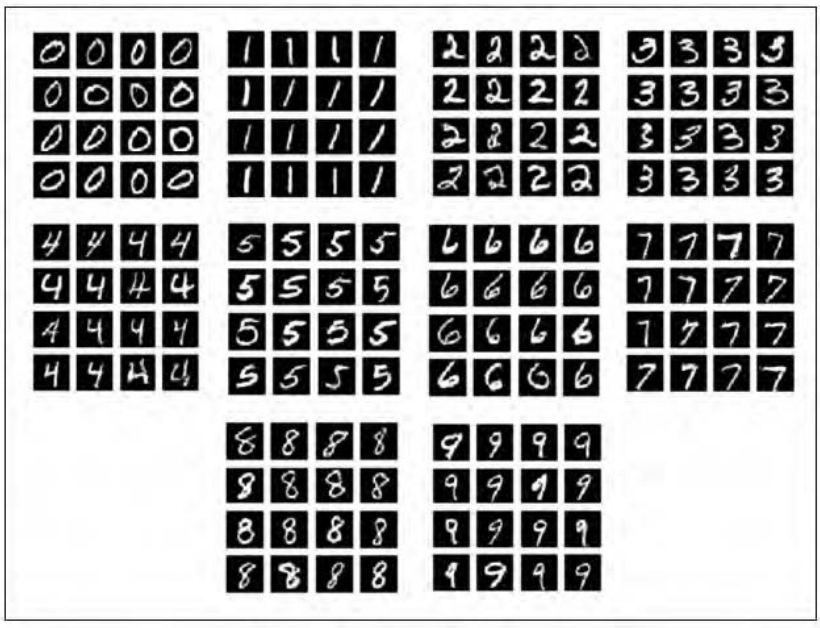

图 10.16: 由 StackedGAN 生成的图像，因为离散代码从 0 到 9 变化。z_0 和 z_1 都是从均值为 0、标准差为 0.5 的正态分布中采样的。

2. 变化的第一个噪声代码，z_0，作为一个恒定的矢量，从-4.0 到 4.0 的数字 0 到 9，如下所示。第二个噪声代码，z_1，被设置为一个零矢量。图10.17显示，第一个噪声代码控制了数字的粗细。例如，对于数字 8。

   python3 stackedgan-mnist-6.2.1.py

   –generator0=stackedgan_mnist-gen0.h5

   –generator1=stackedgan_mnist-gen1.h5 –z0=0 –z1=0 –p0

–digit=8

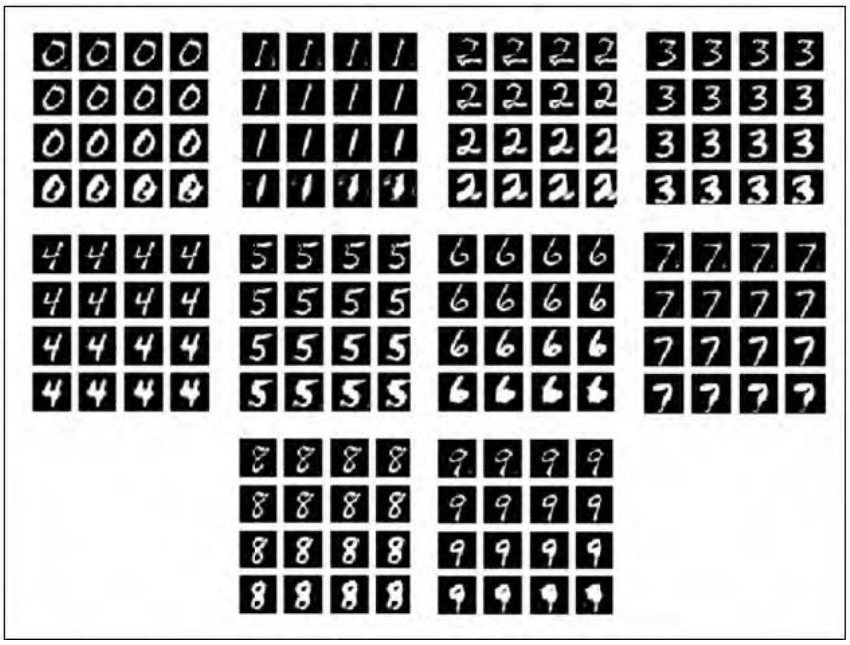

图 10.17: 通过使用 StackedGAN 生成的图像，作为第一个噪声代码，z_0 从一个常数矢量-4.0 到 4.0 变化，用于 0 到 9 位。

3. 变化的第二个噪声代码，z_1，作为一个恒定的矢量，从-1.0 到 1.0 的数字 0 至 9，如下所示。第一个噪声代码，z_0，被设置为一个零矢量。图 6.2.11 显示，第二个噪声代码控制旋转（倾斜），并在一定程度上控制数字的厚度。例如，对于数字 8。

   python3 stackedgan-mnist-6.2.1.py

   –generator0=stackedgan$_mnist - gen0.h5$

–generator1=stackedgan$_mnist - gen1.h5 - -z0 = 0 - -z1 = 0 - -p1$

–digit=8

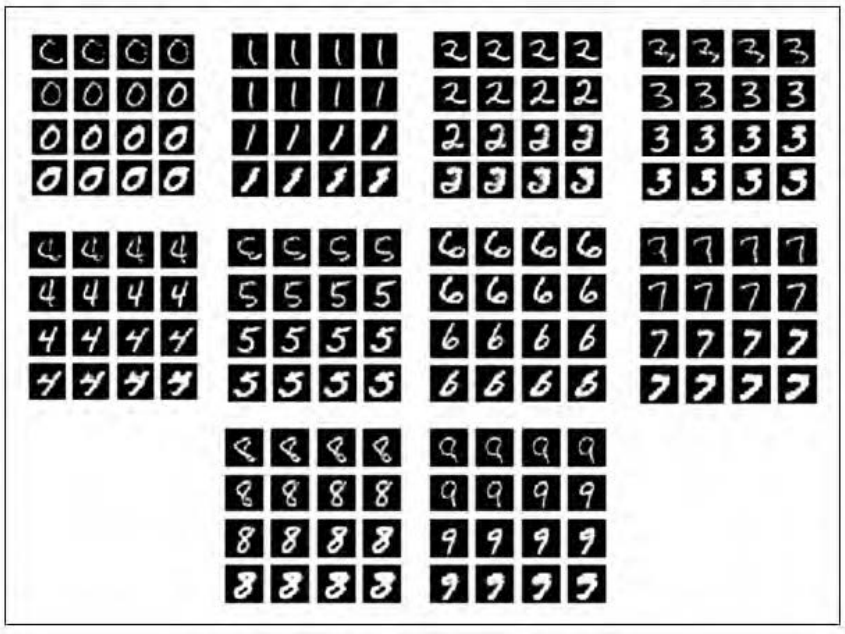

图 10.18: 由 StackedGAN 生成的图像作为第二个噪声代码，z_1，从一个恒定的矢量-1.0 到 1.0 变化，用于数字 0 到 9。z_1 似乎控制着每个数字的旋转（倾斜）和笔画的厚度

图10.16至图10.18表明，StackedGAN 在发生器输出的属性方面提供了额外的控制。控制和属性是（标签，哪个数字）、（z_0，数字的厚度）和（z_1，数字的倾斜度）。从这个例子来看，我们还可以控制其他可能的实验，比如:

- 将堆栈中的元素数量从目前的增加到 2
- 减少代码 z_0 和 z_1 的维度，就像在 InfoGAN 中一样

图10.19显示了 InfoGAN 和 StackedGAN 的潜在代码之间的差异。

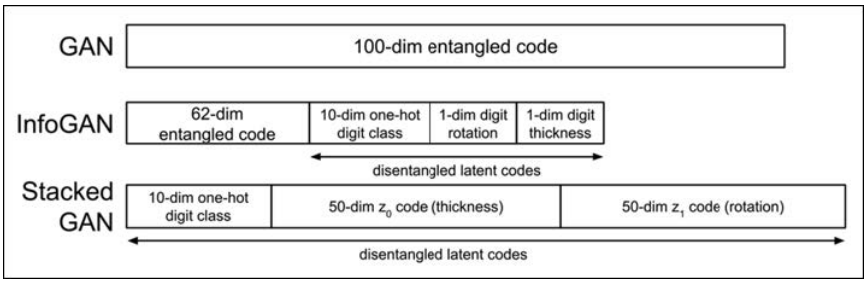

图 10.19: 不同 GANs 的潜势表示

分解代码的基本思路是对损失函数进行约束，使其只受代码的特定属性影响。从结构上看，与 StackedGAN 相比，InfoGAN 更容易实现。InfoGAN 的训练速度也更快。

10.5. 总结

在这一章中，我们已经讨论了如何拆分 GANs 的潜像。在本章的早些时候，我们讨论了 InfoGAN 如何最大化互信息，以迫使生成器学习分解的潜像向量。在 MNIST 数据集的例子中，InfoGAN 使用三个表征和一个噪声代码作为输入。噪声以纠缠表示的形式表示其余的属性。StackedGAN 以不同的方式处理这个问题。它使用一个堆叠的编码器-GANs 来学习如何合成假的特征和图像。编码器首先被训练以提供一个特征数据集。然后，编码器-GANs 被联合训练，学习如何使用噪声代码来控制发生器输出的属性。

10.6. 参考文献

1. Xi Chen et al.: InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets. Advances in Neural Information Processing Systems, 2016

2. Xun Huang et al. Stacked Generative Adversarial Networks. IEEE Conference on Computer Vision and Pattern Recognition (CVPR). Vol. 2, 2017