第 5 章

CHAPTER 5

深度生成模型

深度生成神经网络(**deep generative neural network**)是无监督深度学习模型的主流算法。这类模型旨在学习数据的生成过程。生成模型不仅学习从数据中提取模式,还可估计潜在的概率分布。生成模型用于生成遵循与给定训练集具有相同概率分布的数据。本章将讲解生成模型及其工作方式。

本章将介绍以下实战案例:

- 使用 GAN 生成图像;
- 实现深度卷积 GAN(DCGAN);
- 实现变分自动编码器(VAE)。

5.1 使用 GAN 生成图像

生成对抗网络(**Generative Adversarial Network**,**GAN**)被广泛用于学习数据中潜在的概率分布并生成相同分布的数据集。GAN 由两个网络组成:一个是**生成器**,它可以从正态分布或均匀分布中生成新的数据来构建样本集;另一个是**鉴别器**(也称作评价器或判别器),它可以对生成的样本进行评估并检查真伪,即它们是否属于原始训练数据分布。生成器和鉴别器彼此对抗,生成器相当于伪造者,鉴别器相当于警察,伪造者的目标是通过生成虚假数据来欺骗警察,而警察的作用是检测数据的真伪。来自鉴别器的反馈被传递到生成器,以便它可以在每次迭代时使用。请注意,尽管两个网络都优化了不同且相反的目标函数,但整个系统的稳定性和准确性取决于这两个网络的各自准确性。

以下是 GAN 网络的总体目标函数:

$$\min_G \max_D V(D,G)=E_{x\sim p_{\text{data}}(x)}[\log(D(x))]+E_{z\sim p_z(z)}[\log(1-(D(G(z))))]$$

其中,$G(z)$用于将 z 映射到潜在空间;$D(x)$是从数据空间到概率分数的映射,概率分数表示正确识别真实数据的期望值。

GAN 模型的结构如图 5-1 所示。

本实例使用 GAN 模型实现手写数字生成。

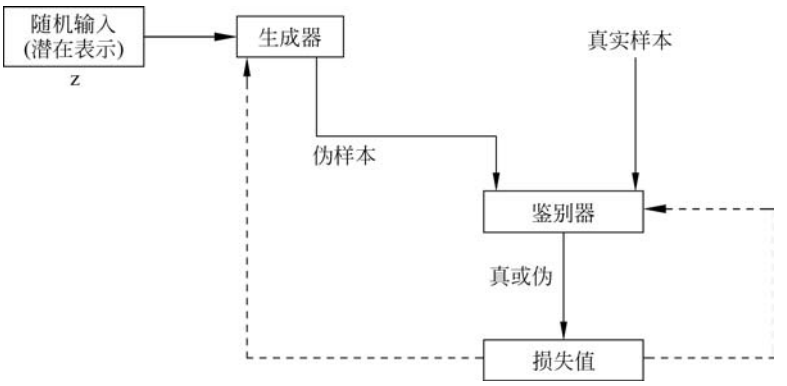


图 5-1　GAN 模型的结构图

5.1.1　准备工作

本实例使用 MNIST 手写数字数据集。它由 60 000 张训练图片和 10 000 张测试灰度图片组成，图片尺寸为 28×28 像素。

首先加载所需的库：

```
library(keras)
library(grid)
library(abind)
```

接着，加载数据集：

```
# 定义输入图像的尺寸变量
img_rows <- 28
img_cols <- 28
# 将数据集的样本随机乱序后分割为训练集和验证集
mnist <- dataset_mnist()
x_train <- mnist$train$x
y_train <- mnist$train$y
x_test <- mnist$test$x
y_test <- mnist$test$y
```

查看数据维度信息：

```
dim(x_train)
```

从图 5-2 可以看到，训练数据中有 60 000 张图片，每个图片的尺寸为 28×28 像素。

```
60000  28  28
```

图 5-2　训练数据的图片数量和尺寸

将训练数据尺寸从 28×28 矩阵重构为包含 784 个元素的一维数组。

```
x_train <- array_reshape(x_train, c(nrow(x_train), 784))
```

将训练数据标准化到 0～1 范围内：

```
x_train <- x_train/255
```

输出并查看一个样本图像数据形式：

```
x_train[1,]
```

现在对数据已有所了解，下面进行模型构建。

5.1.2 操作步骤

GAN 网络包含生成器和鉴别器两部分。首先创建单独的生成器和鉴别器网络，然后通过 GAN 模型将这两个网络连接起来并训练。

(1) 因为处理的是灰度图像，所以通道的数量为 1。同时设置随机噪声向量的维数为 100，随机噪声向量作为生成器网络的输入。

```
channels <- 1
set.seed(10)
latent_dimension <- 100
```

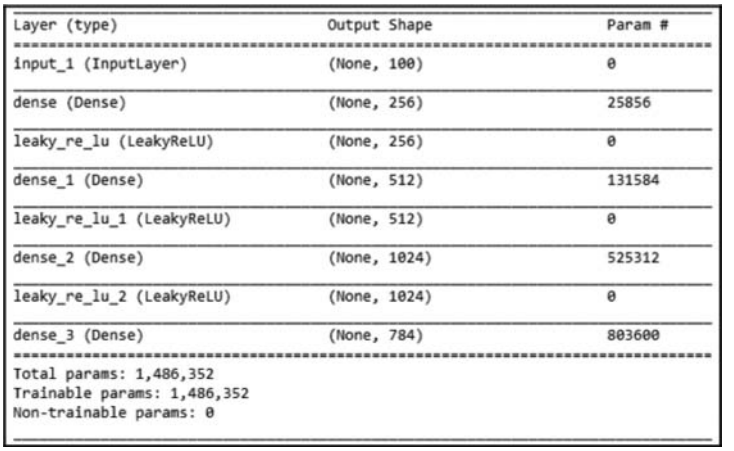
(2) 接下来，创建生成器网络。生成器网络将输入图像数据平坦化后叠加随机噪声向量，噪声向量的维度由 latent_dimension 变量定义。生成器网络由 3 层隐藏层组成，激活函数为 Leaky ReLU。

```
input_generator <- layer_input(shape = c(latent_dimension))
output_generator <- input_generator %>%
 layer_dense(256,input_shape = c(784),kernel_initializer =
initializer_random_normal(mean = 0, stddev = 0.05, seed = NULL))
%>%
 layer_activation_leaky_relu(0.2) %>%
 layer_dense(512) %>%
 layer_activation_leaky_relu(0.2) %>%
 layer_dense(1024) %>%
 layer_activation_leaky_relu(0.2) %>%
 layer_dense(784,activation = "tanh")
generator <- keras_model(input_generator, output_generator)
```

查看生成器网络的摘要信息：

```
summary(generator)
```

生成器网络的摘要信息如图 5-3 所示。



Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 100)	0
dense (Dense)	(None, 256)	25856
leaky_re_lu (LeakyReLU)	(None, 256)	0
dense_1 (Dense)	(None, 512)	131584
leaky_re_lu_1 (LeakyReLU)	(None, 512)	0
dense_2 (Dense)	(None, 1024)	525312
leaky_re_lu_2 (LeakyReLU)	(None, 1024)	0
dense_3 (Dense)	(None, 784)	803600

Total params: 1,486,352
Trainable params: 1,486,352
Non-trainable params: 0

图 5-3　生成器网络的摘要信息

（3）创建鉴别器网络。该网络判定生成器生成的图像为真的概率。

```
input_discriminator <- layer_input(shape = c(784))
output_discriminator <- input_discriminator %>%
 layer_dense(units = 1024,input_shape = c(784),kernel_initializer
= initializer_random_normal(mean = 0, stddev = 0.05, seed = NULL))
%>%
 layer_activation_leaky_relu(0.2) %>%
 layer_dropout(0.3)%>%
 layer_dense(units = 512) %>%
 layer_activation_leaky_relu(0.2) %>%
 layer_dropout(0.3)%>%
 layer_dense(units = 256) %>%
 layer_activation_leaky_relu(0.2) %>%
 layer_dropout(0.3)%>%
 layer_dense(1,activation = "sigmoid")
discriminator <- keras_model(input_discriminator,
output_discriminator)
```

查看鉴别器网络摘要信息：

```
summary(discriminator)
```

鉴别器网络的摘要信息如图 5-4 所示。

完成鉴别器网络配置后，需要进行编译。使用 adam 作为优化器和 binary_crossenropy 作为损失函数。学习率设置为 0.0002。参数 clipvalue 定义梯度裁剪值，它限制了每步迭代中梯度的最大取值，在损失函数的梯度较大的位置效果明显。

```
discriminator %>% compile(
 optimizer = optimizer_adam(lr = 0.0002, beta_1 = 0.5,clipvalue = 1),
```

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 784)	0
dense (Dense)	(None, 1024)	803840
leaky_re_lu (LeakyReLU)	(None, 1024)	0
dropout (Dropout)	(None, 1024)	0
dense_1 (Dense)	(None, 512)	524800
leaky_re_lu_1 (LeakyReLU)	(None, 512)	0
dropout_1 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 256)	131328
leaky_re_lu_2 (LeakyReLU)	(None, 256)	0
dropout_2 (Dropout)	(None, 256)	0
dense_3 (Dense)	(None, 1)	257

Total params: 1,460,225
Trainable params: 1,460,225
Non-trainable params: 0

图 5-4 鉴别器网络的摘要信息

```
 loss = "binary_crossentropy"
)
```

(4) 在开始训练 GAN 网络之前冻结鉴别器网络的权值。这使得鉴别器不可训练,并且在训练 GAN 时它的权重不会更新。

```
freeze_weights(discriminator)
```

(5) 配置 GAN 网络并进行编译。GAN 网络由生成器网络和鉴别器网络组成。

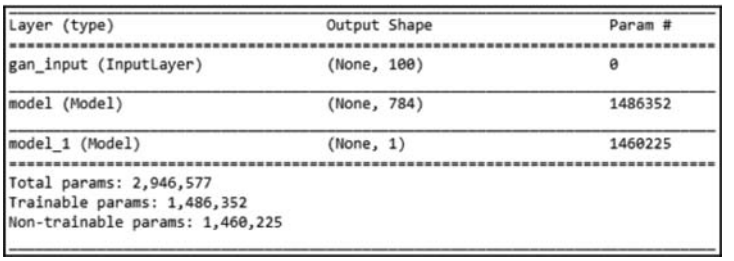
```
gan_input <- layer_input(shape = c(latent_dimension),name = 'gan_input')
gan_output <- discriminator(generator(gan_input))
gan <- keras_model(gan_input, gan_output)
gan %>% compile(
 optimizer = optimizer_adam(lr = 0.0002, beta_1 = 0.5,clipvalue = 1),
 loss = "binary_crossentropy"
)
```

查看 GAN 模型的摘要信息:

```
summary(gan)
```

GAN 模型的摘要信息如图 5-5 所示。

(6) 训练 GAN 网络。设定 GAN 网络的迭代步数为 1000 次,每次迭代生成 20 个新图像,创建一个名为 gan_images 的目录,并在该目录中存储每次迭代生成的图像。每次迭代后将模型参数存储在 gan_model 目录中。



Layer (type)	Output Shape	Param #
gan_input (InputLayer)	(None, 100)	0
model (Model)	(None, 784)	1486352
model_1 (Model)	(None, 1)	1460225

Total params: 2,946,577
Trainable params: 1,486,352
Non-trainable params: 1,460,225

图 5-5　GAN 模型的摘要信息

```
iterations <- 1000
batch_size <- 20
# 创建 gan_images 目录保存生成的图像
dir.create("gan_images")
# 创建 gan_model 目录保存模型训练的参数信息
dir.create("gan_model")
```

开始训练 GAN 网络。

```
start_index <- 1
for (i in 1:iterations) {
# 从正态分布数中随机取 batch_size * latent_dimension 个数,定义 latent_vectors 矩阵
 latent_vectors <- matrix(rnorm(batch_size * latent_dimension),
 nrow = batch_size, ncol = latent_dimension)
# 使用生成器网络将上述随机点生成为假图像
 generated_images <- generator %>% predict(latent_vectors)
# 将假图像与真图像结合起来,作为鉴别器的训练数据
 stop_index <- start_index + batch_size - 1
 real_images <- x_train[start_index:stop_index,]
 rows <- nrow(real_images)
 combined_images <- array(0, dim = c(rows * 2,
dim(real_images)[-1]))
 combined_images[1:rows,] <- generated_images
 combined_images[(rows+1):(rows*2),] <- real_images
 dim(combined_images)
# 为真图像和假图像添加标签
 labels <- rbind(matrix(1, nrow = batch_size, ncol = 1),
 matrix(0, nrow = batch_size, ncol = 1))
# 向标签添加随机噪声以增加鉴别器的鲁棒性
 labels <- labels + (0.5 * array(runif(prod(dim(labels))),
 dim = dim(labels)))
# 使用真假图像训练鉴别器
 discriminator_loss <- discriminator %>%
train_on_batch(combined_images, labels)
# latent_vectors 矩阵采用正态分布数重新初始化
```

```
 latent_vectors <- matrix(rnorm(batch_size * latent_dimension),
 nrow = batch_size, ncol = latent_dimension)

 misleading_targets <- array(0, dim = c(batch_size, 1))
# 使用 GAN 模型训练生成器,注意鉴别器的权重被冻结
 gan_model_loss <- gan %>% train_on_batch(
 latent_vectors,
 misleading_targets
 )
 start_index <- start_index + batch_size
 if (start_index > (nrow(x_train) - batch_size))
 start_index <- 1
# 指定哪些迭代步要保存模型参数和生成的图像
 if(i %in% c(5,10,15,20,40,100,200,500,800,1000)){
 # 保存模型
save_model_hdf5(gan,paste0("gan_model/gan_model_",i,".h5"))
 # 保存生成的图像
 generated_images <- generated_images * 255
 generated_images = array_reshape(generated_images ,dim =
c(batch_size,28,28,1))
 generated_images = (generated_images -min(generated_images
))/(max(generated_images ) -min(generated_images ))
 grid = generated_images[1,,,]
 for(j in seq(2,5)){
 single = generated_images[j,,,]
 grid = abind(grid,single,along = 2)
 }
 png(file = paste0("gan_images/generated_digits_",i,".png"),
 width = 600, height = 350)
 grid.raster(grid, interpolate = FALSE)
 dev.off()
 }
}
```

生成手写数字图像如图 5-6 所示。

图 5-6　模型生成手写数字图像

从图 5-6 可以看出模型运行良好。下面将深入讲解各步骤的原理。

5.1.3 原理解析

5.1.2 节的步骤(1)中,定义了输入图像的尺寸和通道的数量。本实例使用的图像是灰度图,所以指定通道数为 1。步骤(1)还定义了噪声数据维度,作为生成器的输入。在步骤(2)中,构建了一个生成器网络。生成器网络根据 latent_dimension 变量设定的随机噪声向量来生成图像。它生成一个 784 维的输出张量。本实例使用一个深度神经网络作为生成器网络。注意,在生成器的最后一层使用 tanh 作为激活函数,因为它的性能比 sigmoid 激活函数更好。此外,隐藏层中使用 Leaky ReLU 激活函数,因为该激活函数通过允许较小的负激活值来放宽稀疏梯度约束。

建议使用正态分布而不是均匀分布中随机采样生成噪声向量,以获得更好的结果。

5.1.2 节的步骤(3)定义和编译生成器网络。它将生成器生成的大小为 784 的向量映射到一个概率值,该概率值指示生成的图像为真的概率。由于本实例的生成网络是一个有 3 个隐藏层的深度神经网络,所以鉴别器也是一个层数相同的深度神经网络。请注意,在鉴别器的标签中添加了 dropout 层和随机噪声,引入随机性使 GAN 模型具有鲁棒性。在步骤(4)中,冻结了鉴别器的权重,使其不可训练。

在 5.1.2 节的步骤(5)中,配置并编译了 GAN 网络。GAN 网络同时将生成器和鉴别器连接起来。可以将 GAN 网络表示为:

$$gan(x) \leftarrow discriminator(generator(x))$$

创建的 GAN 网络,将生成器生成的图像映射到鉴别器,然后评估图像的真伪。在 5.1.2 节的步骤(6)中,训练了 GAN 网络。为了训练 GAN,需要训练鉴别器,使它能够准确地识别真假图像。生成器使用来自鉴别器的反馈来更新其权值。通过这种方式,鉴别器帮助训练生成器。使用 GAN 模型的损失函数相对于生成器网络权值来求解梯度值用以训练生成器。这样,在每次迭代时,使生成器的权值朝着一个方向移动,使鉴别器更有可能将生成器解码的图像分类为真实的图像。生成器和鉴别器的鲁棒性对整个网络的准确性至关重要。最后,保存每次迭代的模型参数和生成的图像。

5.1.4 内容拓展

尽管 GAN 网络已经成为一种非常流行的深度学习技术,但使用 GAN 网络仍然存在一些挑战。这里列出了一些 GAN 的缺点。

- GAN 非常难训练。通常,模型参数不稳定且不收敛。
- 有时,鉴别器会很准确,以至于生成器有梯度消失问题,无法更新网络。
- 生成器和鉴别器之间的不平衡会导致过拟合。
- GAN 网络对模型调优和超参数选择过于敏感。

5.1.5 参考阅读

要了解更多有关其他类型的GAN模型,请参阅以下链接:

- 条件生成式对抗网络(https://arxiv.org/pdf/1411.1784.pdf)。
- Wasserstein GAN(WGAN)(https://arxiv.org/pdf/1904.08994.pdf,https://arxiv.org/pdf/1704.00028.pdf)。
- 最小二乘GAN(https://arxiv.org/pdf/1611.04076.pdf)。

5.2 实现深度卷积生成对抗网络

卷积GAN模型是一类非常成功的GAN模型。卷积GAN模型在生成器和鉴别器网络中都包含卷积层。本实例将实现一个**深度卷积生成对抗网络(Deep Convolutional Generative Adversarial Network,DCGAN)**。该模型对基础GAN模型(vanilla GAN)进行改进,因为基础GAN模型的结构稳定。遵循以下操作规则有助于提高DCGAN模型的鲁棒性。规则具体如下:

- 在判别器中使用卷积步幅替换池化层,并在生成器网络中使用转置卷积;
- 除了输出层外,在生成器和鉴别器中使用批量规范化(batch normalization);
- 不要使用全连接的隐藏层;
- 在生成器中使用ReLU激活函数,输出层使用tanh激活函数;
- 在鉴别器中使用Leaky ReLU激活函数。

5.2.1 准备工作

本实例将使用花朵识别数据集的部分数据,该数据集由Alexsandr Mamaev创建。本实例使用的数据集包含大约2500张向日葵、蒲公英和雏菊3种花卉的图片。每类花朵由大约800张照片组成。数据集可以从Kaggle网站下载:https://www.kaggle.com/alxmamaev/flowers-recognition

加载所需的库:

```
library(keras)
library(reticulate)
library(abind)
library(grid)
```

现在,可以将数据加载到R环境中。利用Keras库的flow_images_from_directory()函数来加载数据。数据存储于flowers目录中,该目录包含子目录,每个子目录存储一类花朵图像。由于输入的图像尺寸不是统一的,因此在加载图像数据时,指定了图像尺寸,以便每个图像都相应地调整大小。

```
train_path <- "data/flowers/"
image_width = 32
image_height = 32
target_image_size = c(image_width,image_height)
training_data <- flow_images_from_directory(directory =
train_path,target_size = target_image_size, color_mode = "rgb", class_mode
= NULL, batch_size = 2500)
training_data = as_iterator(training_data)
training_data = iter_next(training_data)
training_data <- training_data/255
dim(training_data)
```

```
2500 32 32 3
```

图 5-7 训练数据的图片数量和尺寸

训练数据的维度信息如图 5-7 所示。

现在对于训练数据已有所了解,下面开始模型建立。

5.2.2 操作步骤

先定义模型所需的几个变量。

(1) 根据图片高度、宽度和通道数量来定义图像的尺寸。本实例要对彩色图像进行分析,所以将通道数量保持在 3 个,即 RGB 模式。还要定义噪声向量的尺寸:

```
latent_dim <- 32
height <- 32
width <- 32
channels <- 3
```

(2) 创建生成器网络。生成器网络将具有 latent_dim 个元素的随机向量映射到输入图像,本实例中输入图像的尺寸为(32,32,3)。

```
input_generator <- layer_input(shape = c(latent_dim))
output_generator <- input_generator %>%
# 将输入数据转换为 16x16 像素的 128 - 通道的特征图
 layer_dense(units = 128 * 16 * 16) %>%
 layer_activation_leaky_relu() %>%
 layer_reshape(target_shape = c(16, 16, 128)) %>%
# 添加卷积层
 layer_conv_2d(filters = 256, kernel_size = 5,
 padding = "same") %>%
 layer_activation_leaky_relu() %>%
# 调用 layer_conv_2d_transpose()函数将数据转换为 32×32
 layer_conv_2d_transpose(filters = 256, kernel_size = 4,
 strides = 2, padding = "same") %>%
 layer_activation_leaky_relu() %>%
# 在网络中添加更多卷积层
 layer_conv_2d(filters = 256, kernel_size = 5,
 padding = "same") %>%
```

```
 layer_activation_leaky_relu() %>%
 layer_conv_2d(filters = 256, kernel_size = 5,
 padding = "same") %>%
 layer_activation_leaky_relu() %>%
# 生成 32×32 像素的 1-通道特征图
 layer_conv_2d(filters = channels, kernel_size = 7,
 activation = "tanh", padding = "same")
generator <- keras_model(input_generator, output_generator)
```

查看生成器网络的摘要信息：

```
summary(generator)
```

生成器网络的摘要信息如图 5-8 所示。

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 32)	0
dense (Dense)	(None, 32768)	1081344
leaky_re_lu (LeakyReLU)	(None, 32768)	0
reshape (Reshape)	(None, 16, 16, 128)	0
conv2d (Conv2D)	(None, 16, 16, 256)	819456
leaky_re_lu_1 (LeakyReLU)	(None, 16, 16, 256)	0
conv2d_transpose (Conv2DTranspose)	(None, 32, 32, 256)	1048832
leaky_re_lu_2 (LeakyReLU)	(None, 32, 32, 256)	0
conv2d_1 (Conv2D)	(None, 32, 32, 256)	1638656
leaky_re_lu_3 (LeakyReLU)	(None, 32, 32, 256)	0
conv2d_2 (Conv2D)	(None, 32, 32, 256)	1638656
leaky_re_lu_4 (LeakyReLU)	(None, 32, 32, 256)	0
conv2d_3 (Conv2D)	(None, 32, 32, 3)	37635

Total params: 6,264,579
Trainable params: 6,264,579
Non-trainable params: 0

图 5-8　生成器网络的摘要信息

(3) 创建鉴别器网络。该鉴别器网络将图像映射为尺寸(32,32,3)的二值张量,并估计生成图像为真的概率。

```
input_discriminator <- layer_input(shape = c(height, width,
channels))
output_discriminator <- input_discriminator %>%
 layer_conv_2d(filters = 128, kernel_size = 3) %>%
 layer_activation_leaky_relu() %>%
 layer_conv_2d(filters = 128, kernel_size = 4, strides = 2) %>%
 layer_activation_leaky_relu() %>%
```

```
 layer_conv_2d(filters = 128, kernel_size = 4, strides = 2) %>%
 layer_activation_leaky_relu() %>%
 layer_conv_2d(filters = 128, kernel_size = 4, strides = 2) %>%
 layer_activation_leaky_relu() %>%
 layer_flatten() %>%
 # 添加 dropout 层
 layer_dropout(rate = 0.3) %>%
 # 分类器层(全连接层)
 layer_dense(units = 1, activation = "sigmoid")
discriminator <- keras_model(input_discriminator,
output_discriminator)
```

查看鉴别器模型的摘要信息：

```
summary(discriminator)
```

鉴别器模型的摘要信息如图 5-9 所示。

```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 32, 32, 3)         0
_________________________________________________________________
conv2d (Conv2D)              (None, 30, 30, 128)       3584
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 30, 30, 128)       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 14, 14, 128)       262272
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 14, 14, 128)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 6, 6, 128)         262272
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 6, 6, 128)         0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 2, 2, 128)         262272
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 2, 2, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 512)               0
_________________________________________________________________
dropout (Dropout)            (None, 512)               0
_________________________________________________________________
dense (Dense)                (None, 1)                 513
=================================================================
Total params: 790,913
Trainable params: 790,913
Non-trainable params: 0
```

图 5-9　判别器模型的摘要信息

完成模型配置后，需要编译模型。使用 rmsprop 为优化器、binary_crossentropy 为损失函数，学习率为 0.0008。参数 clipvalue 设定梯度裁剪值，限制迭代中最大和最小梯度值，在损失函数的梯度较陡峭的位置效果明显。

```
discriminator %>% compile(
 optimizer = optimizer_rmsprop(lr = 0.0008,clipvalue = 1.0,decay = 1e-8),
 loss = "binary_crossentropy"
)
```

(4) 在开始训练 GAN 网络之前,先冻结鉴别器的权值,使其不可训练:

```
freeze_weights(discriminator)
```

(5) 配置 DCGAN 网络并编译。GAN 网络由生成器网络和鉴别器网络组成。

```
gan_input <- layer_input(shape = c(latent_dim),name =
'dc_gan_input')
gan_output <- discriminator(generator(gan_input))
gan <- keras_model(gan_input, gan_output)
gan %>% compile(
 optimizer = optimizer_rmsprop(lr = 0.0004,clipvalue = 1.0,decay = 1e-8),
 loss = "binary_crossentropy"
)
```

查看 GAN 模型的摘要信息:

```
summary(gan)
```

GAN 模型的摘要信息如图 5-10 所示。

```
_____________________________________________________________________________
Layer (type)                     Output Shape                  Param #
=============================================================================
dc_gan_input (InputLayer)        (None, 32)                    0
_____________________________________________________________________________
model (Model)                    (None, 32, 32, 3)             6264579
_____________________________________________________________________________
model_1 (Model)                  (None, 1)                     790913
=============================================================================
Total params: 7,055,492
Trainable params: 6,264,579
Non-trainable params: 790,913
```

图 5-10 GAN 模型的摘要信息

(6) 开始训练网络。设置 DCGAN 网络迭代 2000 次,每次迭代生成 40 个新图像。创建一个名为 dcgan_images 的目录,将为每次迭代生成的图像存储在该目录中。将每次迭代时的模型参数存储在 dcgan_model 目录中。

```
iterations <- 2000
batch_size <- 40
dir.create("dcgan_images")
dir.create("dcgan_model")
```

开始模型训练。

```
start_index <- 1
for (i in 1:iterations) {
# 从正态分布数中随机取 batch_size * latent_dimension 个数,定义 latent_vectors 矩阵
 random_latent_vectors <- matrix(rnorm(batch_size * latent_dim),
 nrow = batch_size, ncol = latent_dim)
# 使用生成器网络将上述随机点生成为假图像
```

```
 generated_images <- generator %>% predict(random_latent_vectors)
# 将假图像与真图像结合起来,作为鉴别器的训练数据
 stop_index <- start_index + batch_size - 1
 real_images <- training_data[start_index:stop_index,,,]
 rows <- nrow(real_images)
 combined_images <- array(0, dim = c(rows * 2,
dim(real_images)[-1]))
 combined_images[1:rows,,,] <- generated_images
 combined_images[(rows+1):(rows*2),,,] <- real_images
 # 为真图像和假图像添加标签
 labels <- rbind(matrix(1, nrow = batch_size, ncol = 1),
 matrix(0, nrow = batch_size, ncol = 1))
 # 向标签添加随机噪声以增加鉴别器的鲁棒性
 labels <- labels + (0.5 * array(runif(prod(dim(labels))),
 dim = dim(labels)))
 # 使用真假图像训练鉴别器
 discriminator_loss <- discriminator %>%
train_on_batch(combined_images, labels)
 # latent_vectors 矩阵采用正态分布数重新初始化
 random_latent_vectors <- matrix(rnorm(batch_size * latent_dim),
 nrow = batch_size, ncol = latent_dim)
 misleading_targets <- array(0, dim = c(batch_size, 1))
 # 使用 GAN 模型训练生成器,注意鉴别器的权重被冻结
 gan_model_loss <- gan %>% train_on_batch(
 random_latent_vectors,
 misleading_targets
 )
 start_index <- start_index + batch_size
 if (start_index > (nrow(training_data) - batch_size))
 start_index <- 1
# 指定哪些迭代步要保存模型参数和生成的图像
if(i %in% c(5,10,15,20,40,100,200,500,800,1000,1500,2000)){
# 保存模型
save_model_hdf5(gan,paste0("dcgan_model/gan_model_",i,".h5"))
# 保存生成的图像
 generated_images <- generated_images *255
 generated_images = array_reshape(generated_images ,dim =
c(batch_size,32,32,3))
 generated_images = (generated_images -min(generated_images
))/(max(generated_images )-min(generated_images ))
 grid = generated_images[1,,,]
 for(j in seq(2,5)){
 single = generated_images[j,,,]
 grid = abind(grid,single,along = 2)
 }
```

```
 png(file = paste0("dcgan_images/generated_flowers_",i,".png"),
 width = 600, height = 350)
 grid.raster(grid, interpolate = FALSE)
 dev.off()
 }
}
```

经过 2000 次迭代后,生成图像如图 5-11 所示。

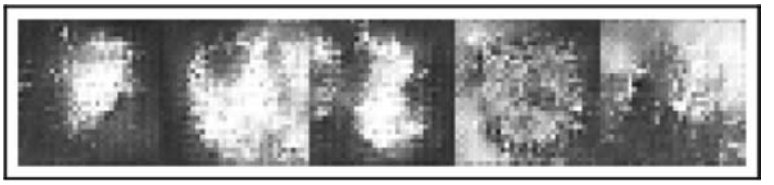


图 5-11　模型 2000 次迭代后生成的图像

如果想要模型更精确,则需要增加迭代步数。

5.2.3　原理解析

在 5.2.2 节的步骤(1)中,定义了输入图像的形状和通道的数量。由于使用的图像是彩色的,指定通道的数量为 3,采用 RGB 模式。步骤(1)还定义了随机噪声的维数。在步骤(2)中,构建了一个生成器网络。生成器网络将具有 latent_dim 个元素的随机向量映射到尺寸为(32,32,3)的图像数据中。

TIP 建议使用正态分布而不是均匀分布中随机采样生成噪声向量,以获得更好的结果。

本实例使用深度卷积网络作为生成器网络。Layer_conv_2d_transpose()函数用于对图像做上采样。在生成器的最后一层使用 tanh 作为激活函数,在隐藏层使用 Leaky Relu 作为激活函数。

TIP 在下采样时,建议使用较大的卷积步幅替代最大池化,以避免出现梯度稀疏的风险。

在 5.2.2 节的步骤(3)中,配置并编译鉴别器网络。鉴别器将生成器生成的尺寸(32,32,3)的图像映射到一个概率值,以表明生成的图像为真的概率。由于生成器网络是卷积网络,所以鉴别器也是卷积网络。为了 GAN 模型添加随机性以提高模型的鲁棒性,在判别器的类标签上添加了 dropout 层和随机噪声。

在 5.2.2 节的步骤(4)中,冻结鉴别器的权重,使其不可训练。在步骤(5)中,配置并编译 GAN 网络。GAN 网络将生成器生成的图像映射到鉴别器获得真图像的概率估计。在最后一步,训练 GAN 网络。在训练 GAN 时,需要训练鉴别器,使其能够准确地识别真伪图像。生成器使用来自鉴别器的反馈来更新其权值。使用 GAN 模型的损失函数相对于生成器网络权值来求解梯度值用以训练生成器。最后,为每次迭代保存模型参数和生成的图像。

5.2.4 内容拓展

尽管 DCGAN 的体系结构是稳定的，但仍然不能保证模型收敛，训练可能是不稳定的。在训练 GAN 时应用一些架构特性和训练过程，模型的性能会显著提高。这些技术利用启发式算法解决收敛问题，改进了模型的学习性能和样本生成。事实上，在一些特定数据集上，GAN 生成的数据与原数据集的真实数据几乎真假难辨，例如 MNIST、CIFAR 数据集等。

以下是一些可以用来提升效果的技巧：

- **特征匹配(feature matching)**。该技术为生成器提供了一个新的目标，使生成的数据与真实数据的统计信息相匹配，而不是直接最大化判别器的输出。标识符用来指定值的匹配的统计信息，并训练生成器匹配标识符中间层特征的期望值。
- **小批量判别(minibatch discrimination)**。与 GAN 相关的一个挑战是生成器总是被一个特定的参数设置所破坏，这使得它总是生成类似的数据。这是因为鉴别器的梯度可能指向许多相似点的相似方向，因为它独立处理每个批次，单个批次之间没有关联。因此，生成器不需要学习如何区分不同批次。小批量判别允许判别器关联地而不是孤立地查看多个样本，从而帮助生成器相应地调整其梯度。
- **历史平均(historical averaging)**。在这种技术中，在更新参数时考虑每个参数过去值的平均值。这种学习方式适用于长时间序列。将生成器和判别器的成本值修改为包含以下项：

$$\left\| \theta - \sum_{i=1}^{t} \theta[i] \right\|^2$$

其中，$\theta[i]$表示参数θ 的过去第 i 个取值。

- **单边标签平滑(one-sided label smoothing)**。这种技术用平滑的值替换分类器的 0 和 1 标签值，例如 0.9 或 0.1，这在处理对抗例子时提高了模型的性能。
- **虚拟批量归一化(virtual batch normalization)**。虽然批量归一化在神经网络中会带来更好的性能，但会导致训练样本的输出依赖于来自同一批次的其他训练样本。虚拟批量归一化通过在训练开始前从固定的参考批次样本计算统计数据，然后对每个训练样本的结果进行归一化来避免这种依赖性。这种技术很耗费计算量，因为前向传播是在两个小批量数据上运行的。因此，这只用于生成器网络。

5.2.5 参考阅读

要了解更多关于使用 ACGAN(Auxiliary Classifier GAN)进行条件图像合成的知识，请读者参阅论文 https://arxiv.org/pdf/1610.09585.pdf。

5.3 实现变分自动编码器

第4章介绍了自动编码器,自动编码器学习输入数据在降维的潜在特征空间中的表示。自动编码器学习一个任意函数来表示输入数据的压缩潜在表示。**一个变分自动编码器**(**Variational AutoEncoder,VAE**),不是学习任意函数,而是学习压缩表示的概率分布的参数。如果可以从这个分布中采样,则可以产生新的数据。VAE由编码器网络和解码器网络组成。

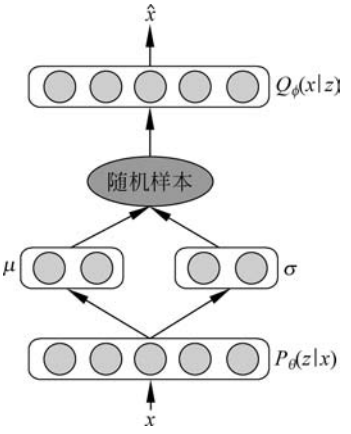


图 5-12 VAE 模型的结构图

VAE模型的结构如图5-12所示。

构成VAE模型的编码器和解码器原理如下:

- **编码器**。编码器是一个神经网络,输入数据用符号 x 表示,输出数据是对输入的潜在表示,记为 z。编码器的功能是求解数据潜在分布的均值(μ)和标准差(σ),生成潜在分布的一个随机样本 z。本质上,VAE编码器学习概率分布 $P_\theta(z|x)$,其中 θ 是编码器网络的参数。
- **解码器**。解码器是从随机样本 z 重建编码器网络的输入数据 x,z 是均值为 μ 和标准差为 σ 的分布的一个随机样本。本质上是得到一个概率分布 $Q_\phi(x|z)$,其中 ϕ 是解码器网络的参数。

在典型的自动编码器中,损失函数由两部分组成:重构损失和一个正则项(惩罚项)。一个训练样本的VAE损失函数如下式所示:

$$l(\theta,\phi)=-E_{z\sim P_\theta(z|x)}[\log(Q_\phi(x\mid z))]+\mathrm{KL}(P_\theta(z\mid x)\mid P(z))$$

方程的第一项为损失值,也就是数据的负对数似然。第二项是学习概率分布 $P_\theta(z|x)$ 和潜在概率分布 $P(z)$ 之间的KL散度。在VAE中,可以假定隐变量的概率分布是标准正态分布,即 $P(z)$ 是 $N(0,1)$。

本实例实现一个变分自动编码器来生成图像。

5.3.1 准备工作

本实例使用MNIST数据集。在第2章中使用过这个数据集,分为训练集和验证集。本实例将每个尺寸为28×28像素的图像重构为一个包含784个元素的数组。

首先导入所需的库:

```
library(keras)
library(abind)
library(grid)
```

加载并重构数据集：

```
mnist <- dataset_fashion_mnist()
x_train <- mnist$train$x/255
x_test <- mnist$test$x/255
x_train <- array_reshape(x_train, c(nrow(x_train), 784), order = "F")
x_test <- array_reshape(x_test, c(nrow(x_test), 784), order = "F")
```

数据准备好后，下面将构建一个 VAE 模型。

5.3.2　操作步骤

本节将通过构建 VAE 模型来重建 MNIST 数据集图像。首先从定义 VAE 的网络参数开始。

(1) 需要定义一些变量来设置网络参数，如样本批量值、输入维数、隐变量维数和迭代次数。

```
# 模型参数
batch_size <- 100L
input_dim <- 784L
latent_dim <- 2L
epochs <- 10
```

(2) 定义 VAE 网络的编码器部分的输入层和隐藏层。

```
input <- layer_input(shape = c(input_dim))
x <- input %>% layer_dense(units = 256, activation = "relu")
```

(3) 配置代表潜在分布的对数标准差和均值的全连接层。

```
# 潜在分布的均值
z_mean <- x %>% layer_dense(units = latent_dim,name = "mean")
# 潜在分布的对数标准差
z_log_sigma <- x %>% layer_dense(units = latent_dim,name = "sigma")
```

(4) 定义一个采样函数，这样就可以从潜在空间中采集新的样本。

```
# 采样函数
sampling <- function(arg) {
 z_mean <- arg[, 1:(latent_dim)]
 z_log_var <- arg[, (latent_dim + 1):(2 * latent_dim)]
 epsilon <- k_random_normal(shape = list(k_shape(z_mean)[1],
latent_dim),
 mean = 0, stddev = 1)
 z_mean + k_exp(z_log_sigma) * epsilon
}
```

(5) 创建一个层,取潜在分布的均值和标准差,并从中生成一个随机样本。

```
# 潜在分布的随机点
z <- layer_concatenate(list(z_mean, z_log_sigma)) %>%
layer_lambda(sampling)
```

(6) 到目前为止,已经定义了一个层来提取一个随机样本。现在,为 VAE 的解码器部分创建一些隐藏层,并将它们组合起来创建输出层。

```
# VAE 解码器的隐藏层
x_1 <- layer_dense(units = 256, activation = "relu")
x_2 <- layer_dense(units = input_dim, activation = "sigmoid")
# 解码器输出
vae_output <- x_2(x_1(z))
```

(7) 构建一个变分自动编码器并可视化输出模型摘要信息:

```
# 变分自动编码器
vae <- keras_model(input, vae_output)
summary(vae)
```

VAE 模型的摘要信息如图 5-13 所示。

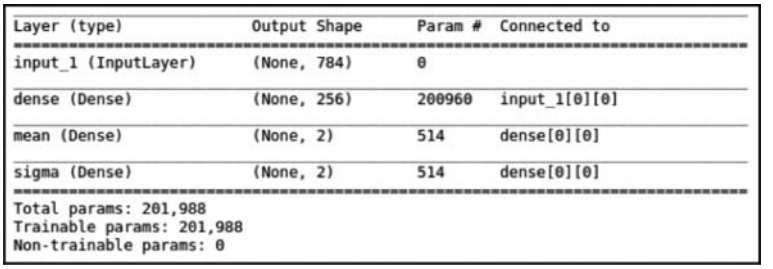
```
Layer (type)                 Output Shape              Param #   Connected to
================================================================================
input_1 (InputLayer)         (None, 784)               0
--------------------------------------------------------------------------------
dense (Dense)                (None, 256)               200960    input_1[0][0]
--------------------------------------------------------------------------------
mean (Dense)                 (None, 2)                 514       dense[0][0]
--------------------------------------------------------------------------------
sigma (Dense)                (None, 2)                 514       dense[0][0]
--------------------------------------------------------------------------------
concatenate (Concatenate)    (None, 4)                 0         mean[0][0]
                                                                 sigma[0][0]
--------------------------------------------------------------------------------
lambda (Lambda)              (None, 2)                 0         concatenate[0][0]
--------------------------------------------------------------------------------
dense_1 (Dense)              (None, 256)               768       lambda[0][0]
--------------------------------------------------------------------------------
dense_2 (Dense)              (None, 784)               201488    dense_1[0][0]
================================================================================
Total params: 404,244
Trainable params: 404,244
Non-trainable params: 0
```

图 5-13 VAE 模型的摘要信息

(8) 创建一个单独的编码器模型:

```
# 创建编码器,将输入映射到潜在空间
encoder <- keras_model(input, c(z_mean,z_log_sigma))
summary(encoder)
```

编码器模型的摘要信息如图 5-14 所示。



Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 784)	0	
dense (Dense)	(None, 256)	200960	input_1[0][0]
mean (Dense)	(None, 2)	514	dense[0][0]
sigma (Dense)	(None, 2)	514	dense[0][0]

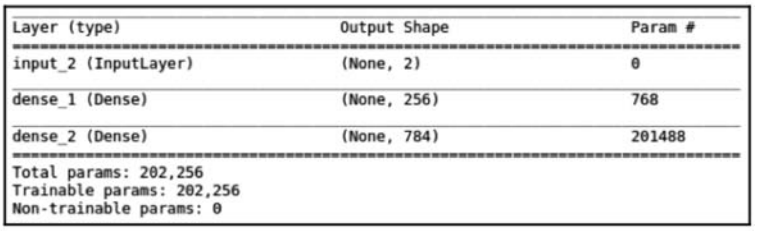
Total params: 201,988
Trainable params: 201,988
Non-trainable params: 0

图 5-14　编码器模型的摘要信息

(9) 创建一个独立的解码器模型：

```
# 解码器的输入层
decoder_input <- layer_input(k_int_shape(z)[-1])
# 解码器的隐藏层
decoder_output <- x_2(x_1(decoder_input))
# 创建解码器
decoder <- keras_model(decoder_input,decoder_output)
summary(decoder)
```

解码器模型的摘要信息如图 5-15 所示。



Layer (type)	Output Shape	Param #
input_2 (InputLayer)	(None, 2)	0
dense_1 (Dense)	(None, 256)	768
dense_2 (Dense)	(None, 784)	201488

Total params: 202,256
Trainable params: 202,256
Non-trainable params: 0

图 5-15　解码器模型的摘要信息

(10) 定义 VAE 模型的损失函数：

```
# 损失函数
vae_loss <- function(x, decoded_output){
 reconstruction_loss <- (input_dim/1.0) * loss_binary_crossentropy(x,
decoded_output)
 kl_loss <- -0.5 * k_mean(1 + z_log_sigma - k_square(z_mean) -
k_exp(z_log_sigma), axis = -1L)
 reconstruction_loss + kl_loss
}
```

(11) 编译模型：

```
vae %>% compile(optimizer = "rmsprop", loss = vae_loss)
```

接着训练模型：

```
vae %>% fit(
 x_train, x_train,
 shuffle = TRUE,
 epochs = epochs,
 batch_size = batch_size,
 validation_data = list(x_test, x_test)
)
```

(12) 查看由模型生成的部分样本图像：

```
random_distribution = array(rnorm(n = 20,mean = 0,sd = 4),dim = c(10,2))
predicted = array_reshape(predict(decoder,matrix(c(0,0),ncol = 2)),dim = c(28,28))
for(i in seq(1,nrow(random_distribution))){
 one_pred = predict(decoder,matrix(random_distribution[i,],ncol = 2))
 predicted = abind(predicted,array_reshape(one_pred,dim =
c(28,28)),along = 2)
}
options(repr.plot.width = 10, repr.plot.height = 1)
grid.raster(predicted,interpolate = FALSE)
```

第 10 次迭代生成的图像如图 5-16 所示。

图 5-16　模型第 10 次迭代生成的图像

下面将详细解释实现步骤的原理。

5.3.3　原理解析

5.3.2 节的步骤(1)定义了自动编码器网络的参数。设置输入数据为 784 个元素的向量，784 是 MNIST 数据集中图像展成的一维向量的元素个数。步骤(2)定义了 VAE 模型的输入层、第一个隐藏层，该隐藏层包括 256 个神经元，激活函数是 ReLU。步骤(3)创建了两个全连接层：z_mean 和 z_sigma。这两层的神经元数量等于潜在分布的维度。本实例中，将 784 维的输入数据压缩表示为二维的潜在空间。注意，这些层分别与它们的前一层进行全连接。这些层表示的是均值为 μ 和标准偏差为 σ 的潜在分布。步骤(4)定义了一个抽样函数，它从一个均值和方差已知的分布中产生一个随机样本。它以一个四维张量作为输入，从张量中提取均值和标准偏差，并从该分布中生成一个随机样本。根据生成新的随机样本 $\mu+\sigma(\varepsilon)$，其中 ε 是服从标准正态分布的随机值。

在 5.3.2 节的步骤(5)中，创建了一个层来连接 z_mean 和 z_sigma 层的输出张量，然后堆叠到一个 Lambda 层。Keras 中的 Lambda 层是一个包装器，它将任意表达式包装为一

个神经网络层(用户自定义的层)。本实例中 Lambda 层封装了在上一步中定义的抽样函数。这一层的输出是 VAE 解码器的输入。步骤(6)构建了 VAE 的解码器网络。实例化了两个层,x_1 和 x_2,分别有 256 和 784 个神经元。将这些层组合起来创建输出层。步骤(7)创建了 VAE 模型。

在 5.3.2 节的步骤(8)和步骤(9)中,分别创建了编码器和解码器模型。步骤(10)定义了 VAE 模型的损失函数。它是重构损失值加上 Kullback-Leibler 散度值,Kullback-Leibler 散度值通过隐变量的假设真实概率分布形式和输入数据条件下隐变量出现的条件概率分布来求解。步骤(11)编译了 VAE 模型,并使用 rmsprop 优化器对其进行了 10 次迭代训练,以使 VAE 损失函数最小化。在最后一步中,生成了一个新的合成图像样本。

5.3.4 参考阅读

要了解更多关于自然语言处理的生成模型,请查看以下链接:

- GPT-2: https://openai.com/blog/better-language-models/
- BERT: https://arxiv.org/pdf/1810.04805.pdf