chapter 5：深度学习：循环神经网络RNN和长短期记忆网络LSTM

目录

本章是对第四章的内容的扩展，讨论了循环神经网络RNN和长短期记忆神经网络LSTM。虽然本章的大部分内容为关于这些模型的说明，但也有基于Keras的代码示例。因此，建议您先阅读附录中Keras相关的材料。

本章的第一部分介绍了RNN、BPTT（延时反向传播算法）的体系结构，并包含一个简短的基于Keras的代码示例。正如您所见，RNN可以跟踪早期信息，这使得它在多种任务中都很有用，包括NLP任务。

本章的第二部分介绍了LSTM的体系结构，它比RNN更复杂。具体来说，低成本存储器包括一个遗忘门forget gate、一个输入门input gate、一个输出门output gate，以及一个长期存储单元。您还将了解LSTM相对于RNN的优势。此外，您将接触到一些常见的NLP相关模型中使用双向LSTM的例子。

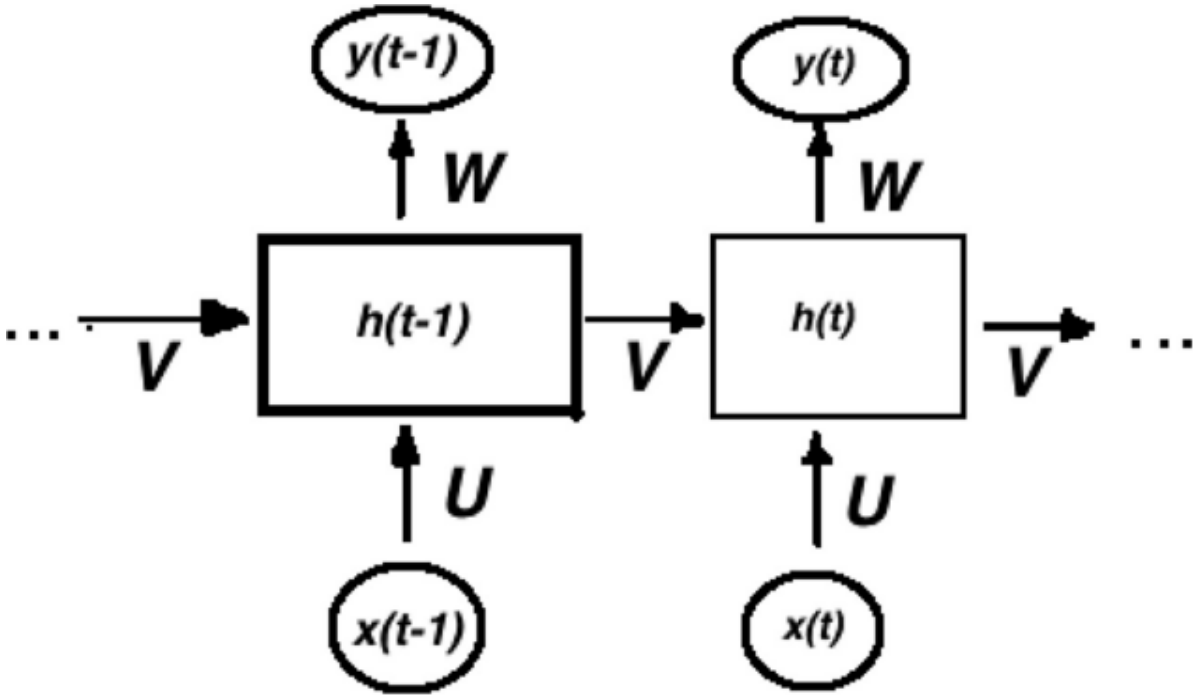
本章的第三部分介绍了自动编码器的体系结构及其基本原理，同时对各种自动编码器进行了介绍。

请记住，理解本章的代码示例需要对Keras有一定的了解。

什么是RNN?

RNN指循环神经网络，是一种在20世纪80年代开发的体系结构。RNN适用于处理包含顺序数据的数据集，也适用于自然语言处理任务，如语言建模、文本生成或句子自动完成。事实上，RNN还可以进行图像分类（如MINIST）。图5.1展示了一个简单的RNN的内容。

图5.1 RNN示例



除了简单的RNN，还发展出了更强大的结构，如LSTM和GRU。基本RNN具有最简单类型的反馈机制（稍后会提到），并且包括Sigmoid激活函数。

RNN（包括LSTM和GRU）和ANN的几个重要的不同之处在于如下几个方面：

- 有状态性（所有RNN都有）
- 反馈机制（所有RNN都有）
- Sigmoid或tanh激活函数
- 多门（LSTM和GRU具有）
- BPTT（延时间反向传播）
- 截断BPTT

首先，ANN和CNN本质上是无状态的，而RNN是有状态的，因为其存在内部状态。因此，RNN能够处理更复杂的输入序列，这使得它适用于手写识别和语音识别等任务。

RNN的解剖

考虑图5.1中的RNN。假设输入序列记为x1,x2,x3,...,x(t),...，隐藏的状态序列记为h1,h2,h3,...,h(t)。注意，每个输入序列和隐藏的状态序列都是1×n的向量，其中n是特征的数量。

在时间周期t内，输入是基于h(t-1)和x(t)的组合，之后将激活函数应用于该组合（也可以添加偏置向量）。

另一个区别是在连续时间段发生的RNN的反馈机制。具体来说，将前一时间段的输出和当前时间段的新的输入相结合，来计算新的内部状态。我们表示用序列{h(0),h(1),h(2),...h(t-1),h(t)}来表示一个RNN的内部状态在时间段{0,1,2,...,t-1,t}之内的序列，我们假设序列{x(0),x(1),x(2),...,x(t-1),x(t)}是相同时间段的输入序列。

RNN在时间周期t内的基本关系如下：

$$h(t) = f(W * x(t) + U * h(t-1))$$

在上述公式中，W和U是权重矩阵，f通常是tanh激活函数。

下面是一个基于TF 2 Keras的模型的代码片段，它基于tf.keras.layers.SimpleRNN类：

```
import tensorflow as tf
...
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.SimpleRNN(5, input_shape=(1,2), batch_input_shape=
[1,1,2],      stateful=True))
...
```

在线搜索，了解更多关于Keras和RNN的信息和代码示例。

什么是BPTT?

RNN中的BPTT（延时间反向传播算法）与CNN的反向传播是对等的。为了训练神经网络，在BPTT中更新RNN的权重矩阵。

然而，在RNN中可能出现梯度爆炸的问题，也就是说，梯度变得任意大（相当于在消失梯度场景中梯度变得任意小）。处理梯度爆炸问题的一种方法是使用截断BPTT，这意味着BPTT只在少数步骤中使用，而不是所有步骤。另一种方法是指定梯度的最大值，这用到简单的条件逻辑。

一个好消息是，有另一种方法可以克服梯度爆炸和梯度消失问题，用到LSTM，随后将在本章讨论。

使用RNN和Keras

Listing5.1展示了keras_rnn_model.py的内容，说明了如何创建一个简单的基于Keras的RNN模型。

Listing 5.1: keras_rnn_model_py

```
import tensorflow as tf

timesteps = 30
input_dim = 12

# number of units in RNN cell
units = 512

#number of classes to be identified
n_classes = 5

#Keras Sequential model with RNN and Dense layer
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.SimpleRNN(units=units,dropout=0.2,
                                    input_shape=(timesteps,input_dim)))
model.add(tf.keras.layers.Dense(n_classes,
                                activation='softmax'))

#model loss function and optimizer
model.compile(loss='categorical_crossentropy',
              optimizer=tf.keras.optimizers.Adam(),
              metrics=['accuracy'])

model.summary()
```

Listing5.1首先初始化变量timesteps（时间步长数）、input_Dim（每个输入向量的维度）、units（RNN神经元中的隐藏单元数）和n_classes（数据集中的类数）。

Listing5.1的下一部分创建了一个基于Keras的模型，看起来类似于早期的基于Keras的模型，除了RNN层的代码片段，如下所示：

```
model.add(tf.keras.layers.SimpleRNN(units=units,
                                    dropout=0.2,
                                    input_shape=(timesteps, input_dim)))
```

如你所见，上述代码片段添加了SimpleRNN类的实例以及前面代码块中定义的变量。

代码的最后一部分调用compile()，然后调用summary()来显示模型的结构。

执行Listing5.1中的代码，你将看到如下输出：

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
Shape
=================================================================
simple_rnn (SimpleRNN)       (None, 512)               268800
_________________________________________________________________
dense (Dense)                (None, 5)                 2565
=================================================================
Total params: 271,365
Trainable params: 271,365
Non-trainable params: 0
```

现在你已经看到了在Keras创建基于RNN的模型是多么简单，让我们来看一个在Keras创建的基于RNN模型的例子，它将在MNIST数据集上进行训练，这是下一节的主题。

使用Keras、RNN和MNIST

Listing5.2展示了keras_rnn_mnist.py的内容，说明了如何创建一个简单的基于Keras的RNN模型，该模型在MNIST数据集上进行训练。

Listing5.2：keras_rnn_mnist.py

```
#Simple RNN and MNIST dataset
import tensorflow as tf
import numpy as np

# instantiate mnist and load data:
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# one-hot encoding for all labels to create 1x10
# vectors that are compared with the final layer:
y_train = tf.keras.utils.to_categorical(y_train)
y_test  = tf.keras.utils.to_categorical(y_test)

#resize and normalize the 28x28 images:
image_size = x_train.shape[1]
x_train = np.reshape(x_train,[-1, image_size, image_size])
x_test  = np.reshape(x_test, [-1, image_size, image_size])
x_train = x_train.astype('float32') / 255
x_test  = x_test.astype('float32')  / 255

#initialize some hyper-parameters:
input_shape = (image_Size, image_size)
batch_Size = 128
```

```
hidden_units = 128
dropout_rate = 0.3

#RNN-based Keras model with 128 hidden units:
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.SimpleRNN(units=hidden_units,
                                    dropout=dropout_rate,
                                    input_shape=input_shape))
model.add(tf.keras.layers.Dense(num_labels))
model.add(tf.keras.layers.Actication('softmax'))
model.summary()

model.compile(loss='categotical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])

#train the network on the training data:
model.fit(x_train, y_train, epochs=8, batch_size=batch_size)

#calculate and then display the accuracy:
loss, acc = model.evaluate(x_test, y_test, batch_size=batch_size)
print("\nTest accuracy: %.1f%%" % (100.0 * acc))
```

Listing5.2包括import语句，随后初始化mnist变量作为对MNIST数据集的引用，之后初始化用于训练数据和测试数据的四个变量。

Listing5.2的下一部分确保训练图像和测试图像被调整为28×28图像，之后图像的像素值（在0到255范围内）被按比例缩小到0到1。Listing5.2的下一部分与Listing5.1非常相似：初始化了一些超参数，然后在Keras中创建了一个基于RNN的模型。

现在我们有了新的代码块，首先将模型结构保存在rnn-mnist.png文件中。第二个新代码块调用compile()将模型与训练数据同步，然后调用fit()训练模型。

Listing5.2的最后一部分根据测试数据评估训练好的模型，并展示loss和acc值，分别对应于测试数据上模型的损失和精确度。执行Listing5.2的代码，将看到如下输出：

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
simple_rnn (SimpleRNN)	(None, 256)	72960
dense (Dense)	(None, 10)	2570
activation (Activation)	(None, 10)	0

```
Total params: 75,530
Trainable params: 75,530
Non-trainable params: 0

Epoch 1/5
60000/60000 [==============================] - 33s
   542us/sample - loss: 0.8198 - accuracy: 0.7605
```

```
Epoch 2/5
 6528/60000 [==>...........................] - ETA:
   27s - loss: 0.4661 - accuracy: 0.8627
60000/60000 [==============================] - 34s
   559us/sample - loss: 0.3724 - accuracy: 0.8917
Epoch 3/5
60000/60000 [==============================] - 33s
   545us/sample - loss: 0.2764 - accuracy: 0.9183
Epoch 4/5
60000/60000 [==============================] - 33s
   545us/sample - loss: 0.2269 - accuracy: 0.9327
Epoch 5/5
60000/60000 [==============================] - 34s
   561us/sample - loss: 0.1983 - accuracy: 0.9407
10000/10000 [==============================] - 2s
   237us/sample - loss: 0.1396 - accuracy: 0.9577
Test accuracy: 95.8%
```

使用TensorFlow和RNN（可选）

本节中的代码示例是可选的，因为它基于TensorFlow 1.x。在本书出版后，谷歌发布了TensorFlow 2，之后TensorFlow 1.x成为遗留代码，再支持一年的使用。当在本书中遇到任何涉及TensorFlow 1.x的代码示例时，请记住这一点。

然而，这个代码示例确实提供了一些关于RNN神经元中每个隐藏层的输入和状态的底层细节，这可以让你对如何执行计算以及生成值有所了解。请记住，两个时间步骤的数据是模拟的，也就是说，数据没有反映任何有意义的用例。简化数据的目的是帮助你关注计算的执行方式。

Listing5.3展示了dynamic_rnn_2TP.py的内容，说明了如何创建一个简单的基于TensorFlow的RNN模型。

```
import tensorflow as tf
import numpy as np
n_steps = 2        # number of time steps
n_inputs = 3       # number of inputs per time unit
n_neurons = 5      # number of hidden units
X_batch = np.array([
    #t = 0        t = 1
    [[0, 1, 2], [9, 8, 7]],  #instance 0
    [[3, 4, 5], [0 ,0 ,0]],  #instance 1
    [[6, 7, 8], [6, 5, 4]],
    [[9, 0, 1], [3, 2, 1]],
])
#sequence_length <= #of elements in each batch
seq_length_batch = np.array([2, 1, 2, 2])

X = tf.placeholder(dtype=tf.float32, shape=[None, n_steps, n_inputs])
seq_length = tf.placeholder(tf.int32, [None])

basic_cell = tf.nn.rnn_cell.BasicRNNCell(num_units=n_neurons)
outputs, states = tf.nn.dynamic_rnn(basic_cell, X,
                                    sequenct_length=seq_length, dtype=tf.float32)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
```

```
    outputs_val, states_val = sess.run([outputs, states],
                                       feed_dict={X:X_batch,
                                                  seq_length:seq_length_batch})
    print("X_batch     shape:", X_batch.shape) #(4,2,3)
    print("outputs_val shape:", outputs_val.shape) #(4,2,5)
    print("states_val  shape:", states_val.shape)  #(4,5)
    print("outputs_val:",outputs_val)
    print("---------------------\n")
    print("states_val: ",states_val)
################################################################################
    ########################################
# outputs => output of ALL RNN states
# states  => output of LAST ACTUAL RNN state(ignores zero vector)
# state = output[1] for full sequences
# state = output[0] for short sequences
################################################################################
    ########################################
```

Listing5.3首先将n_steps（时间步长数）、n_inputs（输入数）、n_neurons（神经元数）分别初始化为2、3和5。

接下来，NumPy数组X_batch是一个用整数初始化的4×2×3数组。从注释行可以看出，第一列值代表时间步长为0，第二列值代表时间步长为1。你也可以将X_batch中的每一行数据视为两个时间步长的数据实例。

变量seq_length_batch是一个一维整型向量，每个整数指出现在纯零值向量左侧的时间步长数。如你所见，该向量包含实例号0、2和3的值2，以及实例号1的值0。

Listing5.3的下一部分定义了占位符X，它可以容纳任意数量的形状为[n_steps, n_inputs]的数组。现在我们准备定义一个RNN单元，并指定它的输出和状态，如下所示：

```
basic_cell = tf.nn.rnn_cell.BasicRNNCell(num_units=n_neurons)
outputs, states = tf.nn.dynamic_rnn(basic_cell, X,
                                    sequenct_length=seq_length, dtype=tf.float32)
```

要记住的关键点是，最右边隐藏单元的最终输出值是传递给下一个神经元的值。

执行Listing5.3的代码，你将看到如下输出，其中有趣的值以粗体显示。

```
#----------------------------
#outputs_val:
#[[[-0.09700205  0.7671716   0.6775758   0.01522888
   0.5460828 ]
#  [ 0.92776424 -0.5916748   0.67824966  0.99423325
   0.9999991 ]]
#
# [[ 0.24040672  0.81568515  0.8890421   0.780813
   0.99762475]
#  [ 0.          0.          0.          0.        0.
          ]]
#
# [[ 0.5282535   0.8549201   0.9647311   0.9692446
   0.99999046]
#  [ 0.9725177  -0.7165484   0.46688017  0.9411293
   0.9999323 ]]
#
# [[ 0.81080747 -0.9926888   0.56612366  0.9561879
   0.9997731 ]
#  [ 0.48786768 -0.7099759  -0.7283263   0.76442945
   0.9971904 ]]]
#----------------------------
#states_val:
#[[ 0.92776424 -0.5916748   0.67824966  0.99423325
   0.9999991 ]
# [ 0.24040672  0.81568515  0.8890421   0.780813
   0.99762475]
# [ 0.9725177  -0.7165484   0.46688017  0.9411293
   0.9999323 ]
# [ 0.48786768 -0.7099759  -0.7283263   0.76442945
   0.9971904 ]]
#----------------------------
```

在上述输出中，请注意粗体显示的行数是2、1、2、2，这与seq_length_batch中的值完全相同。如你所见，这些突出显示的行出现在标记为states_val的数组中（也以粗体显示）。

Listing5.3是一个很简单的人工制造的RNN的例子，希望这个例子能让你更好地理解RNN内部的工作方式。RNN有许多变体，可以在以下网址中了解：

https://en.wikipedia.org/wiki/Recurrent_neural_network

什么是LSTM?

LSTM是RNN的一种特殊类型，适合许多任务，包括自然语言处理、语音识别和手写识别。LSTM非常适合处理长期依赖性问题，长期依赖性是指相关信息和需要该信息的位置之间的距离。当文档的一个部分中的信息需要链接到文档中较远位置的信息时，就会出现这种情况。

LSTM开发于1997年，并不断超越最先进的算法的精度性能。LSTM也开始了语音识别革新（大约在2007年）。在2009年，LSTM赢得了模式识别比赛，2014年百度使用RNN打破了语音识别的记录。从以下链接查看LSTM的示例：https://commons.wikimedia.org/w/index.php?curid=60149410

LSTM的解剖

LSTM是有状态的，它包括三个门（forget gate,input gate和output gate），涉及sigmod函数，还包含一个使用到tanh激活函数的单元状态。在时间周期t内，LSTM的输入基于向量h(t-1)和x(t)的组合。输入向量h(t-1)和x(t)被组合起来，sigmoid函数通过forget gate、input gate和output gate被应用于该组合（也可以包括偏置向量）。

LSTM的短期记忆发生在时间步长t内。LSTM的内部单元保持长期记忆。更新内部单元状态需要用到tanh激活功能，而三个门使用sigmoid激活功能。这里有一个基于TF 2的代码片段，基于Kera定义了LSTM模型。

```
import tensorflow  as tf
...
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.LSTMCell(6,batch_input_shape=
(1,1,1),kernal_initializer='ones',stateful=True))
model.add(tf.keras.layers.Dense(1))
```

你可以通过以下链接来了解LSTM和LSTMCell的区别：https://stackoverflow.com/questions/48187283/whats-the-difference-between-lstm-and-lstmcell

如果你感兴趣，以下链接有关于线性扫描模块以及如何自定义LSTM单元的更多信息：

https://en.wikipedia.org/wiki/Recurrent_neural_network

https://stackoverflow.com/questions/54231440/define-custom-lstm-cell-in-keras

双向LSTM

除了单向的LSTM，你也可以定义一个双向的LSTM，它由两个常规的LSTM组成：一个LSTM向前，一个LSTM向后或相反的方向。你可能会惊讶地发现，双向LSTM非常适合解决NLP任务。

例如，ELMo是使用双向LSTM的NLP任务的深层单词表示。NLP世界中一个新的架构叫transformer，双向transformer被用于BERT。BERT是一个非常知名的系统（由谷歌在2018年发布），可以解决复杂的NLP问题。

下面的TF 2代码块是一个基于Keras的双向LSTM模型：

```
import tensorflow as tf
...
model = Sequential()
model.add(Bidirectional(LSTM(10,return_sequences=True),input_shape=(5,10)))
model.add(Bidirectional(LSTM(10)))
model.add(Dense(5))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy',optimizer='rmsprop')
...
```

前面的代码片段包含两个双向LSTM单元，都以粗体显示。

LSTM公式

LSTM的公式比简单RNN的更新公式更诉法，但有一些模型可以帮助你理解这些公式。从以下链接查看LSTM公式：[https://en.wikipedia.org/wiki/Long_short-term_memory#cite_note-lstm1997-1]

这些公式展示了如何在时间步长t内计算forget gate f、input gate i和output gate o的新权重。此外，上面的链接展示了如何在时间步长t内如何计算新的内部状态和隐藏状态。

注意门f、i和o的模式：它们都计算两项的和，每项都包含x(t)和h(t)的乘积，之后sigmoid函数作用于和。具体来说，下面是t时刻forget gate公式：

$$f(t) = sigma(W(f) * x(t) + U(f) * h(t) + b(f))$$

在上述公式中，W(f)、U(f)和b(f)分别是与x(t)相关的权重矩阵、与h(t)相关的权重矩阵和forget gate f的偏置向量。

请注意，i(t)和o(t)的计算模式与f(t)的计算模式相同。不同的是，i(t)有矩阵W(i)和U(i)，而o(t)有矩阵W(o)和U(o)。因此，f(t)、i(t)和o(t)具有平行结构。

c(t)、i(t)和h(t)的计算基于f(t)、i(t)和o(t)，如下所示：

$$c(t) = f(t) * c(t-1) + i(t) * tanh(c'(t))$$

$$c'(t) = sigma(W(c) * x(t) + U(c) * h(t-1))$$

$$h(t) = o(t) * tanh(c(t))$$

LSTM的最终状态是一个一维向量，包含了LSTM所有其它层的输入。如果你有一个模型包含多个LSTM的模型，则当前LSTM的最终状态向量将成为下一个LSTM的输入。

LSTM超参数调优

LSTM也容易出现过拟合，如果你手动优化LSTM的超参数，这里有一个需要考虑的事项列表：

- 过度拟合（使用正则化，如L1或L2）
- 更大的网络更容易过拟合
- 更多的数据有助于减少过拟合
- 对网络进行多轮训练
- 堆叠层可能有帮助
- 对于LSTM，使用softsign，而不是softmax
- RMSprop、AdaGrad或动量是不错的选择
- 泽维尔权重初始化

通过在线搜索获取更多信息。

使用TensorFlow和LSTM（可选）

Listing5.4展示了dynamic_lstm_2TP.py的内容，说明了如何用TensorFlow 1.x创建一个简单的LSTM模型。

Listing5.4 dynamic_lstm_2TP.py

```
import tensorflow as tf
import numpy as np
n_steps = 2 # number of time steps
n_inputs = 3 # number of inputs per time unit
n_neurons = 5 # number of hidden units
X_batch = np.array([
 # t = 0      t = 1
 [[0, 1, 2], [9, 8, 7]], # instance 0
 [[3, 4, 5], [0, 0, 0]], # instance 1
 [[6, 7, 8], [6, 5, 4]], # instance 2
 [[9, 0, 1], [3, 2, 1]], # instance 3
```

```
])
seq_length_batch = np.array([2, 1, 2, 2])
X = tf.placeholder(dtype=tf.float32,shape=[None,n_steps,n_inputs])
seq_length = tf.placeholder(tf.int32, [None])
basic_cell = tf.nn.rnn_cell.BasicLSTMCell(num_units=n_neurons)
outputs, states = tf.nn.dynamic_rnn(basic_cell, X,
                                    sequence_length=seq_length,
dtype=tf.float32)
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    outputs_val, states_val = sess.run([outputs, states],
    feed_dict={X:X_batch, seq_length:seq_length_batch})
    print("X_batch shape:", X_batch.shape) # (4,2,3)
    print("outputs_val shape:", outputs_val.shape) # (4,2,5)
    print("states: ", states_val) # LSTMStateTuple(...)
    print("outputs_val:",outputs_val)
    print("----------------------------\n")
    print("states_val: ",states_val)
```

Listing5.4的前半部分与Listing5.3的前半部分相同，不同的是第一行代码包括将basic_cell定义为LSTM（以粗体显示），如下所示：

```
  basic_cell = tf.nn.rnn_cell.BasicLSTMCell(num_units=n_neurons)
  outputs, states = tf.nn.dynamic_rnn(basic_cell, X,
                                      sequence_length=seq_length,
dtype=tf.float32)
```

请注意，Listing5.4中的输出和状态的初始化方式与Listing5.3中的完全相同。代码的下一部分是一个作为训练循环的tf.Session()代码块。

Listing5.4中需要注意的另一个区别是：在训练循环的每次计算中，states_val实际上是LSTMStatesTuple的一个实例，而Listing5.3中的states_val是一个4×5的张量。执行Listing5.4中的代码，你将看到如下输出：

```
  ('X_batch shape:', (4, 2, 3))
  ('outputs_val shape:', (4, 2, 5))

  ('states: ', LSTMStateTuple(c=array(
   [[-1.0492262 , -0.1059267 , -0.27163735, -0.64399946, 0.06018598],
   [-0.7445494 , 0.00723887, -0.11805946, -0.26550752, 0.21816696],
   [-1.4126835 , 0.05187892, -0.07408151, -0.66379607, 0.1348486 ],
   [-0.5987958 , 0.24536057, -0.16916996, -0.8177415 , 0.39747238]],
   dtype=float32), h=array(
   [[-7.33636796e-01, -6.07701950e-02, -1.40444040e-01, -2.65002381e-02,
     5.37334010e-04],
   [-4.83454257e-01, 3.39480606e-03, -3.36034223e-02, -2.59866733e-02,
4.49425131e-02],
   [-7.36429453e-01, 2.63450593e-02, -4.42487188e-02, -1.05846934e-01,
5.22684120e-03],
   [-3.73311013e-01, 1.35892674e-01, -9.72046256e-02, -2.79455721e-01,
5.36275432e-02]],
   dtype=float32)))

  ('outputs_val:', array([
   [[-1.39581457e-01, -8.17378387e-02, -8.70967656e-02, -3.05497926e-02,
     1.16406225e-01],
```

```
    [-7.33636796e-01, -6.07701950e-02, -1.40444040e-01, -2.65002381e-02,
     5.37334010e-04]],
    [[-4.83454257e-01, 3.39480606e-03, -3.36034223e-02, -2.59866733e-02,
4.49425131e-02],
    [ 0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
0.00000000e+00]],
    [[-6.21303201e-01, 4.13885061e-03, -6.17417134e-03, -8.89408588e-03,
4.83810157e-03],
    [-7.36429453e-01, 2.63450593e-02, -4.42487188e-02,  -1.05846934e-01,
     5.22684120e-03]],
     [[-1.01410240e-01, 4.99857590e-02, -9.47358180e-03,  -3.74739647e-01,
       9.64458846e-03],
    [-3.73311013e-01, 1.35892674e-01, -9.72046256e-02, -2.79455721e-01,
     5.36275432e-02]]],
   dtype=float32))

   ----------------------------
   ('states_val: ', LSTMStateTuple(c=array(
    [[-1.0492262 , -0.1059267 , -0.27163735, -0.64399946, 0.06018598],
    [-0.7445494 , 0.00723887, -0.11805946, -0.26550752, 0.21816696],
    [-1.4126835 , 0.05187892, -0.07408151, -0.66379607, 0.1348486 ],
    [-0.5987958 , 0.24536057, -0.16916996, -0.8177415 , 0.39747238]],
    dtype=float32), h=array(
    [[-7.33636796e-01, -6.07701950e-02, -1.40444040e-01, -2.65002381e-02,
       5.37334010e-04],
    [-4.83454257e-01, 3.39480606e-03, -3.36034223e-02, -2.59866733e-02,
4.49425131e-02],
    [-7.36429453e-01, 2.63450593e-02, -4.42487188e-02, -1.05846934e-01,
5.22684120e-03],
    [-3.73311013e-01, 1.35892674e-01, -9.72046256e-02, -2.79455721e-01,
5.36275432e-02]],
   dtype=float32)))
```

关于输出，有两点需要特别注意。首先，检查前面输出中以粗体显示的中间部分，注意这些值与最后输出块中显示的值相同，输出部分标记为states_val。

第二，以粗体显示的第二个代码块中包含两个向量：一个非零向量后跟着一个零向量，对应于Listing5.4中标记为instance 1的数据。

什么是GRU?

GRU（门控循环单元）是一种RNN，是LSTM的简化类型。GRU和LSTM的主要区别是，GRU有两个门（复位门reset gate和更新门update gate），而LSTM有三个门（复位门reset gate、输出门output gate和遗忘门forget gate）。GRU中的reset gate执行LSTM中的input gate和forget gate的功能。

请记住，GRU和LSTM都有有效跟踪长期依赖关系的目标，并且它们都解决了梯度消失和梯度爆炸问题。从以下链接查看GRU示例：https://commons.wikimedia.org/wiki/File:Gated_Recurrent_Unit,_base_type.svg

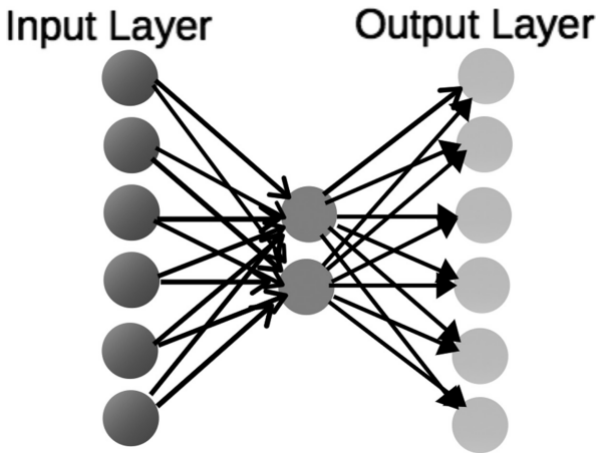
以下链接查看GRU的公式（类似于LSTM的公式）：https://en.wikipedia.org/wiki/Gated_recurrent_unit

什么是自动编码器（Autoencoders）？

自动编码器（AE）是一种类似于MLP的神经网络，其输出层与输入层相同。最简单的自动编码器类型包含一个隐藏层，其神经元数量少于输入层和输出层。然而，有许多不同类型的含有多个隐藏层的自动编码器，有时包含比输入层更多的神经元（有时包含更少的神经元）。

自动编码器使用无监督学习和反向传播来学习有效的数据编码，目的是降维：自动编码器将输入值设置为等于输入，然后尝试找到恒等式。图5.2展示了一个包含简单的自动编码器，它包含一个隐藏层。

图5.2 基本自动编码器



本质上，自动编码器将输入压缩成比输入数据维数少的"中间"向量，然后将其转换成与输入形状相同的张量。下面列出了自动编码器的几个使用案例：

- 文档检索
- 分类
- 异常检测
- 对立自动编码器
- 图像去噪（生成清晰的图像）

  下面是一个使用基于TensorFlow和Keras的自动编码器来执行欺诈检测的一个案例：
  https://www.datascience.com/blog/fraud-detection-with-tensorflow

自动编码器也可用于特征提取，因为它们可以产生比主成分分析法更好的结果。请记住，自动编码器是基于特定数据的，这意味着它们仅适用于相似的数据。然而，它们在图像压缩并非如此（并且在数据压缩方面表现一般）。例如，一个基于人脸的自动编码器在处理树木图片时表现很差。总之，自动编码器包括：

- 将输入"挤压"到更小的层
- 学习一组数据的表示
- 通常用于降维
- 只保留中间的"压缩"层

举一个更高级的例子，考虑一个10x10的图像（100像素），和一个有100个神经元（10x10像素）的AE，一个有50个神经元的隐藏层，和一个有100个神经元的输入层。因此，AE将100个神经元压缩为50个神经元。

如你之前所见，基本自动编码器有许多变体，其中一些如下所列：

- LSTM自动编码器
- 去噪自动编码器
- 收缩式自动编码器
- 堆叠式自动编码器
- 深度自动编码器
- 线性自动编码器

如果你感兴趣，以下链接包含各种自动编码器，包含本节所提到的：https://www.google.com/search?sa=X&q=Autoencoder&tbm=isch&source=univ&ved=2ahUKEwjo8zRrIniAhUGup4KHVgvC10QiR56BAgMEBY&biw=967&bih=672

在线搜索代码示例和关于自动编码器及其相关用例的更多详细信息。

自动编码器和主成分分析PCA

如果自动编码器涉及线性激活或仅涉及单个sigmoid隐藏层，则其最佳解决方案与主成分分析密切相关。

具有大小为p（其中p小于输入的大小）的单个隐藏的的自动编码器的权重跨越与前p个主分量所跨越的向量子空间相同的向量子空间。

自动编码器的输出是这个子空间的正交投影。自动编码器的权重不等于主分量，并且通常不正交，但是可以使用奇异值分解从其中恢复主分量。

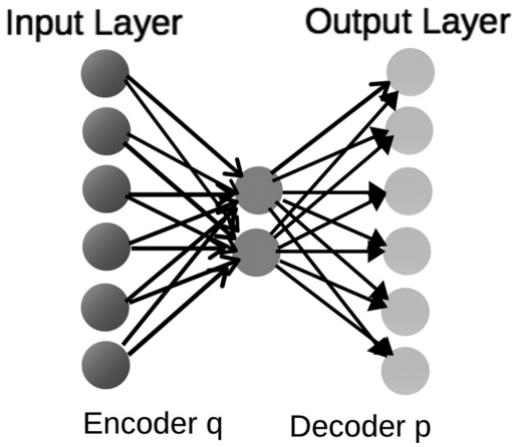
什么是可变自动编码器?

简而言之，可变自动编码器（VAE）是一种增强的常规自动编码器，其中左侧充当编码器，右侧充当解码器。两者都有一个与编码和解码过程相关的概率分布。

另外，编码器和解码器其实都是神经网络。编码器的输入是数值向量x，输出时具有权重和偏移量的隐藏表示z。解码器具有输入a（即编码器的输出），其输出时数据概率分布的参数，也具有权重和偏移量。注意，编码器和解码器的概率分布是不同的。如果你想了解更多关于可变自动编码器的信息，请浏览以下链接：https://en.wikipedia.org/wiki/Autoencoder#Variational_autoencoder_.28VAE.29

图5.3显示了一个简化的自动高级编码器，包含一个单一的隐藏层。

图5.3 可变自动编码器



另一个有趣的模型是CNN和VAE的结合，可以参考以下链接：https://towardsdatascience.com/gans-vs-autoencoders-comparison-of-deep-generative-models-985cf15936ea

在下一节，你讲了解GAN以及如何将VAE和GAN结合起来。

什么是GAN（生成式对抗网络）？

GAN指生成式对抗网络，其最初目的是产生合成数据，通常用于扩充小数据集或不平衡数据集。一个与失踪人员有关的用例：将这些人的可用照片提供给一个GAN，以便生成这些人现在样貌的图像。GAN还有很多其他用例，这里列举一些：

- 生成艺术
- 创造时尚风格
- 改善低质量图像
- 创造“人造”面孔
- 重建不完整或损坏的图像

伊恩·古德菲勒（蒙特拉大学机器学习博士）于2014年创建了GAN。闫恩·勒坤（Facebook的人工智能研究主管）称，对抗训练是"过去十年里最有趣的想法"。顺便说一句，闫恩·勒坤是2019年图灵奖的三位获得者之一，另外两位是约书亚·本吉奥和杰弗里·欣顿。

GAN变得越来越普遍，人们正在为它们寻找创造性的（想不到吧？）用途。可惜的是，GAN已经被用于邪恶的目的，比如自图像识别系统以来的规避。GAN可以通过改变像素值，从有效图像生成伪造图像，以欺骗神经网络。由于这些系统以来像素模式，它们可能会被像素值被改变了的图像——对抗性图像所欺骗。

从以下链接查看一个使用GAN扭曲后的熊猫形象的示例：**https://arxiv.org/pdf/1412.6572.pdf**

这里有一篇文章深入探讨了敌对例子的细节（包括错误分类的熊猫）：**https://openai.com/blog/adversarial-example-research/**

麻省理工学院的一篇论文显示，触发错误分类的修改值利用了图像系统与特定对象相关联的精确模式。研究人员注意到，数据集包含两种类型的相关性：与数据集数据相关的模式，以及数据集数据中不可一般化的模式。GAN成功地利用后一种相关性来欺骗图像识别系统。麻省理工学院的论文详情如下：**https://gandissect.csail.mit.edu**

对抗性的攻击能被阻止吗?

不幸的是，对抗性攻击没有长远的解决办法，鉴于其性质，可能永远无法完全防御它们。尽管目前有各种正在开发的技术来抵御对抗性攻击，但这些技术的有效性往往是短暂的：新的GAN被创造出来，以智胜这些技术。下面的文章包含了关于对抗性攻击的更多信息：https://www.technologyreview.com/s/613170/emtech-digital-dawn-song-adversarial-machine-learning

有趣的是，GAN在收敛方面可能会有问题，就像其他神经网络一样。解决这一问题的一种技术成为迷你批次鉴别，其细节如下：https://www.inference.vc/understanding-minibatch-discrimination-in-gans/

请注意，上述链接涉及到库尔巴克·赛布勒发散和JS发散，这是更高级的话题。前一篇博文还包含以下Jupyter Notebook的链接：**https://gist.github.com/fhuszar/a91c7d0672036335c1783d02c3a3dfe5**

如果你有兴趣使用GAN，这个GitHub链接包含构建攻击和防御的Python和TensorFlow代码示例：**https://github.com/tensorflow/cleverhans**

创建GAN

GAN有两个主要部分：生成器和鉴别器。生成器可以具有类似CNN的结构以产生图像，而鉴别器可以具有类似CNN的结构以便检测图像（由发生器提供）的真假。以此类推，生成器类似于制造假币的人，鉴别器类似于试图区分有效货币和假币的执法人员。

生成器（已经初始化）发送假图像到鉴别器（已经被训练但不再可更新）进行分析。如果鉴别器在检测真图像和假图像方面准确度很高，那么生成器需要被修改以提高其产生的假图像的质量。对生成器的修改是通过反向误差传播来实现的。另一方面，如果鉴别器的性能很差，则说明生成器正在生成高质量的假图像，因此生成器不需要进行大的修改。

Listing5.5展示了keras_create_gan.py的内容，它定义了一个用于创建GAN的python函数。

Listing5.5 keras_create_gan.py

```
import tensorflow as tf
def build_generator(img_shape, z_dim):
    model = tf.keras.models.Sequential()
    # Fully connected layer
    model.add(tf.keras.layers.Dense(128, input_dim=z_dim))
```

```
    # Leaky ReLU activation
    model.add(tf.keras.layers.LeakyReLU(alpha=0.01))
    # Output layer with tanh activation
    model.add(tf.keras.layers.Dense(28 * 28 * 1, activation='tanh'))
    # Reshape the Generator output to image dimensions
    model.add(tf.keras.layers.Reshape(img_shape))
    return model
def build_discriminator(img_shape):
    model = tf.keras.models.Sequential()
    # Flatten the input image
    model.add(tf.keras.layers.Flatten(input_shape=img_shape))
    # Fully connected layer
    model.add(tf.keras.layers.Dense(128))
    # Leaky ReLU activation
    model.add(tf.keras.layers.LeakyReLU(alpha=0.01))
    # Output layer with sigmoid activation
    model.add(tf.keras.layers.Dense(1,activation='sigmoid'))
    return model
def build_gan(generator, discriminator):
    # ensure that the discriminator is not trainable
    discriminator.trainable = False
    # the GAN connects the generator and descriminator
    gan = tf.keras.models.Sequential()
    # start with the generator:
    gan.add(generator)
    # then add the discriminator:
    gan.add(discriminator)
    # compile gan
    opt = tf.keras.optimizers.Adam(lr=0.0002,beta_1=0.5)
    gan.compile(loss='binary_crossentropy',optimizer=opt)
    return gan
gen = build_generator(...)
dis = build_discriminator(...)
gan = build_gan(gen, dis)
```

如你所见，Listing5.5中的Python函数包含三个Python方法，分别用于build_generator()、build_discriminator()和build_gan()，用于创建生成器、鉴别器和GAN。

GAN由一个发生器和一个鉴别器来进行初始化，两者都是初始化函数的参数。请注意，build_gan()中的鉴别器是不可训练的，这可以通过下面的代码片段来验证：

```
discriminator.trainable = False
```

另外需要注意的一点是，前面的Python函数没有创建类似CNN的体系结构。下面的代码块显示了创建鉴别器的不同方法（省略了细节）：

```
dis = build_discriminator(...)
gen_model = tf.keras.models.Sequential()
gen_model.add(tf.keras.layers.Dense(...)
gen_model.add(tf.keras.layers.LeakyReLU(alpha=0.2))
gen_model.add(tf.keras.layers.Reshape(...)
# code for upsampling
gen_model.add(tf.keras.layers.Conv2DTranspose(...)
gen_model.add(tf.keras.layers.LeakyReLU(...)
...
gen_model.add(tf.keras.layers.Reshape(...)
gen_model.add(tf.keras.layers.LeakyReLU(...)
# output layer
gen_model.add(tf.keras.layers.Conv2D(...))
```

上面的代码块涉及Conv2D() 类和LeakyReLU() 类（类似于ReLU），但是请注意没有最大池层。查看在线文档了解关于上采样的解释，以及TensorFlow/Keras类LeakyReLU()和Conv2DTranspose()的用途。

GAN的高层视图

GAN有许多类型，例如DCGAN（深层集合GAN）、cGAN（条件GAN）和StyleGAN。一般来说，创建GAN包括以下高级步骤。

- 步骤1）选择数据集（如MINST或 cifar10）
- 步骤2）定义和训练鉴别器模型
- 步骤3）定义和使用生成器模型
- 步骤4）训练生成器模型
- 步骤5）评估GAN模型的性能
- 步骤6）使用最终的生成器模型

虽然GAN可以类似于CNN，但在它们使用的层中存在一些重要的区别。首先，GAN中的卷积层往往具有(2,2)的步幅，也就是说卷积滤波器一次移动两列，然后一次下移两行。其次，GAN包含一个LeakyReLU激活函数，它与ReLU激活函数略有不同。第三，GAN没有最大缓冲层。

此外，GAN还涉及向上扩展的概念，这在某种意义上类似于向下扩展（即最大池）的对立面。进行在线搜索，了解更多关于GAN的详细信息。

VAE-GAN模型

另一个有趣的模型是VAE-GAN模型，它是VAE和GAN的混合体，关于这个模型的细节如下：https://towardsdatascience.com/gans-vs-autoencoders-comparison-of-deep-generative-models-985cf15936ea

根据上述链接的内容，GAN优于VAE，但GAN也很难使用，需要大量的数据和调整。如果你感兴趣，这里有一个GAN教程：https://github.com/mrdragonbear/GAN-Tutorial

小结

在本章中，我们学习了RNN的体系结构，基于它的有状态性，我们可以解决一些任务。接下来是一个基于Keras的代码示例。然后我们了解了LSTM的体系结构，学习了一个基本的代码示例。

此外，我们还学习了一个基于TensorFlow 1.x的LSTM单元代码示例，它的输出显示了执行的一些内部计算的路径。我们还学习了可变自动编码器，了解了一些用例。

最后，我们学习了GAN，了解了一些关于如何构建GAN的高级描述，以及GAN是如何训练的。