Exercise 1

使用有状态RNN和使用无状态RNN的优缺点是什么？

答案：

无状态 RNN 只能捕获长度小于或等于 RNN 训练窗口大小的模式。相反，有状态 RNN 可以捕获长期模式。然而，实现有状态的 RNN 要困难得多——尤其是正确准备数据集。此外，有状态 RNN 并不总是工作得更好，部分原因是连续批次不是独立同分布 (IID)。梯度下降不喜欢非 IID 数据集。

Exercise 2

为什么人们要使用 encoder–decoder RNNs，而不是简单的 sequence-to-sequence RNNs 来进行自动翻译？

答案：

一般来说，如果你一个字一个字地翻译一个句子，结果会很糟糕。例如，法语句子“Je vous en prie”的意思是“You are welcome”，但如果一次翻译一个词，就会得到“I you in pray”。

最好先阅读整个句子然后再翻译。普通的 sequence-to-sequence RNN 会在读取第一个单词后立即开始翻译句子，而 encoder-decoder RNN 会先读取整个句子然后再进行翻译。也就是说，可以想象一个简单的 sequence-to-sequence RNN ，它会在不确定接下来要说什么时输出静音（就像人类翻译人员在必须翻译直播时所做的那样）。

Exercise 3

您可以如何处理可变长度的输入序列？那么可变长度的输出序列呢？

答案：

可以通过填充较短的序列来处理可变长度的输入序列，以便批处理中的所有序列具有相同的长度，并使用掩码来确保 RNN 忽略填充标记。为了获得更好的性能，您可能还想创建包含相似大小序列的批次。参差不齐的张量可以保存可变长度的序列，Keras 现在支持它们，这简化了对可变长度输入序列的处理（尽管在撰写本文时，它仍然无法将参差不齐的张量作为 GPU 上的目标进行处理）。

对于可变长度的输出序列，如果输出序列的长度是预先知道的（例如，如果你知道它与输入序列相同），那么你只需要配置损失函数，使其忽略在序列结束后出现的标记。同样，将使用该模型的代码应该忽略序列末尾之外的标记。但通常输出序列的长度是事先不知道的，因此解决方案是训练模型，使其在每个序列的末尾输出一个序列结束标记。

Exercise 4

什么是集束搜索，你为什么要使用它？你可以用什么工具来实现它呢？

答案：

集束搜索是一种用于提高经过训练的编码器-解码器模型性能的技术，例如在神经机器翻译系统中。该算法跟踪 k 个最有希望的输出句子的简短列表（例如，前三个），并且在解码器的每个步骤中，它都会尝试将它们扩展一个词；然后它只保留 k 个最有可能的句子。参数 k 称为集束宽度：它越大，使用的 CPU 和 RAM 就越多，但系统的精度也越高。这种技术不是在每一步都贪婪地选择最有可能的下一个词来扩展单个句子，而是允许系统同时探索几个有希望的句子。此外，这种技术非常适合并行化。您可以通过编写自定义存储单元来实现集束搜索。或者，TensorFlow Addons 的 seq2seq API 提供了一个实现。

Exercise 5

什么是注意力机制？它有什么帮助？

答案：

注意力机制是一种最初用于编码器-解码器模型的技术，使解码器能够更直接地访问输入序列，从而使其能够处理更长的输入序列。在每个解码器 time step，当前解码器的状态和编码器的完整输出由对齐模型处理，该模型为每个输入 time step 输出对齐分数。该分数表示输入的哪一部分与当前解码器 time step 最相关。然后将编码器输出的加权和（由它们的对齐分数加权）馈送到解码器，解码器产生下一个解码器状态和该 time step 的输出。使用注意力机制的主要好处是编码器-解码器模型可以成功处理更长的输入序列。另一个好处是对齐分数使模型更易于调试和解释：例如，如果模型出错，您可以查看它关注的是输入的哪一部分，这有助于诊断问题。注意力机制也是 Transformer 架构的核心，在多头注意力层中。请参阅下一个答案。

Exercise 6

transformer 体系结构中最重要的一层是什么？它的目的是什么？

答案：

Transformer 架构中最重要的层是 Multi-Head Attention 层（原始 Transformer 架构包含 18 个，其中包括 6 个 Masked Multi-Head Attention 层）。它是 BERT 和 GPT-2 等语言模型的核心。其目的是让模型识别哪些词彼此最一致，然后使用这些上下文线索改进每个词的表示。

Exercise 7

你什么时候需要使用采样的 softmax？

答案：

当有很多类别（例如，数千）时训练分类模型时使用采样 softmax。它根据模型为正确类别预测的 logit 和错误单词样本的预测 logit 计算交叉熵损失的近似值。与计算所有 logits 的 softmax 然后估计交叉熵损失相比，这大大加快了训练速度。训练完成后，模型可以正常使用，使用常规的 softmax 函数根据所有 logits 计算所有类概率。

Exercise 8

Hochreiter 和 Schmidhuber 在他们关于 LSTM 的论文中使用了嵌入式 Reber 语法。它们是生成诸如"BPBTSXXVPSEPE"之类的字符串的人工语法。查看 Jenny Orr 对该主题的精彩介绍，然后选择特定的嵌入式 Reber 语法（例如 Orr 页面上展示的那个），然后训练 RNN 来识别字符串是否遵循该语法。您首先需要编写一个函数，该函数能够生成包含大约 50% 符合语法的字符串和 50% 不符合语法的训练批次。

答案：

首先，我们需要构建一个基于语法生成字符串的函数。语法将表示为每个状态的可能转换列表。转换指定要输出的字符串（或生成它的语法）和下一个状态。

In [3]:
```
default_reber_grammar = [
    [("B", 1)],           # (state 0) =B=>(state 1)
    [("T", 2), ("P", 3)], # (state 1) =T=>(state 2) or =P=>(state 3)
    [("S", 2), ("X", 4)], # (state 2) =S=>(state 2) or =X=>(state 4)
    [("T", 3), ("V", 5)], # and so on...
    [("X", 3), ("S", 6)],
    [("P", 4), ("V", 6)],
    [("E", None)]]        # (state 6) =E=>(terminal state)

embedded_reber_grammar = [
    [("B", 1)],
    [("T", 2), ("P", 3)],
    [(default_reber_grammar, 4)],
    [(default_reber_grammar, 5)],
    [("T", 6)],
    [("P", 6)],
    [("E", None)]]

def generate_string(grammar):
    state = 0
    output = []
    while state is not None:
        index = np.random.randint(len(grammar[state]))
        production, state = grammar[state][index]
        if isinstance(production, list):
            production = generate_string(grammar=production)
        output.append(production)
    return "".join(output)
```

让我们根据默认的 Reber 语法生成一些字符串：

In [5]:
```
import numpy as np

np.random.seed(42)

for _ in range(25):
    print(generate_string(default_reber_grammar), end=" ")
```

```
BTXXTTVPXTVPXTTVPSE BPVPSE BTXSE BPVVE BPVVE BTSXSE BPTVPXTTTVVE BPVVE BTXSE BTXXVPSE BPTTTTTTTTVVE BTXSE BPVPSE BTXSE BPTVPSE BTXXTVPSE BPVVE BPVVE BPVVE BPTTVVE BPVVE BPVVE BTXXVVE BTXXVVE BTXXVPXVVE
```

看起来不错。现在让我们根据嵌入的 Reber 语法生成一些字符串：

In [6]:
```
np.random.seed(42)

for _ in range(25):
    print(generate_string(embedded_reber_grammar), end=" ")
```

```
BTBPTTTVPXTVPXTTVPSETE BPBPTVPSEPE BPBPVVEPE BPBPVPXVVEPE BPBTXXTTTTVVEPE BPBPVPSEPE BPBTXXVPSEPE BPBTSSSSSSSXSEPE BTBPVVETE BPBTXXVVEPE BPBTXXVPSEPE BTBTXXVVETE BPBPVVEPE BPBPVVEPE BPBTSXSEPE BPBPVVEPE BPBPTVPSEPE BPBTXXVVEPE BTBPTVPXVVETE BTBPVVETE BTBTSSSSSSSXXVVETE BPBTSSSXXTTTTVPSEPE BTBPTTVVETE BPBTXXTVVEPE BTBTXSETE
```

好的，现在我们需要一个函数来生成不符合语法的字符串。我们可以生成一个随机字符串，但这个任务有点太简单了，所以我们将生成一个符合语法的字符串，我们将通过仅更改一个字符来破坏它：

In [7]:
```
POSSIBLE_CHARS = "BEPSTVX"

def generate_corrupted_string(grammar, chars=POSSIBLE_CHARS):
    good_string = generate_string(grammar)
    index = np.random.randint(len(good_string))
    good_char = good_string[index]
    bad_char = np.random.choice(sorted(set(chars) - set(good_char)))
    return good_string[:index] + bad_char + good_string[index + 1:]
```

让我们看一些损坏的字符串：

In [8]:
```
np.random.seed(42)

for _ in range(25):
    print(generate_corrupted_string(embedded_reber_grammar), end=" ")
```

```
BTBPTTTPPXTVPXTTVPSETE BPBTXEEPE BPBPTVVVEPE BPBTSSSSXSETE BPTTXSEPE BTBPVPXTTTTTTEVETE BPBTXXSVEPE BSBPTTVPSETE BPBXVVEPE BEBTXSETE BPBPVPSXPE BTBPVVVETE BPBTSXSETE BPBPTTTPTTTTTVPSEPE BTBTXXTTSTVPSETE BBBTXSETE BPBTPXSEPE BPBPVPXTTTTVPXTVPXVPXTTTVVEVE BTBXXXTVPSETE BEBTSSSSSXXVPXTVVETE BTBXTTVVETE BPBTXSTPE BTBTXXTTTVPSBTE BTBTXSETX BTBTSXSSTE
```

我们不能将字符串直接输入 RNN，因此我们需要以某种方式对它们进行编码。一种选择是对每个字符进行独热编码。另一种选择是使用嵌入。让我们选择第二个选项（但由于只有少数字符，one-hot 编码可能也是一个不错的选择）。为了使嵌入起作用，我们需要将每个字符串转换为字符 ID 序列。让我们为此编写一个函数，使用每个字符在可能字符“BEPSTVX”的字符串中的索引：

In [9]:
```
def string_to_ids(s, chars=POSSIBLE_CHARS):
    return [chars.index(c) for c in s]
```

In [12]:
```
string_to_ids("BTTTXXVVETE")
```

Out[12]:
```
[0, 4, 4, 4, 6, 6, 5, 5, 1, 4, 1]
```

我们现在可以生成数据集，其中包含 50% 的好字符串和 50% 的坏字符串：

In [15]:
```
import tensorflow as tf

def generate_dataset(size):
    good_strings = [
        string_to_ids(generate_string(embedded_reber_grammar))
        for _ in range(size // 2)
    ]

    bad_strings = [
        string_to_ids(generate_corrupted_string(embedded_reber_grammar))
        for _ in range(size - size // 2)
    ]

    all_strings = good_strings + bad_strings

    X = tf.ragged.constant(all_strings, ragged_rank=1)
    y = np.array([[1.] for _ in range(len(good_strings))] +
                 [[0.] for _ in range(len(bad_strings))])

    return X, y
```

In [16]:
```
np.random.seed(42)

X_train, y_train = generate_dataset(10000)
X_valid, y_valid = generate_dataset(2000)
```

让我们看一下第一个训练序列：

In [17]:
```
X_train[0]
```

Out[17]:
```
<tf.Tensor: shape=(22,), dtype=int32, numpy=array([0, 4, 0, 2, 4, 4, 4, 5, 2, 6,
4, 5, 2, 6, 4, 4, 5, 2, 3, 1, 4, 1])>
```

In [18]:
```
y_train[0]
```

Out[18]:
```
array([1.])
```

完美的！我们已准备好创建 RNN 来识别好的字符串。我们构建一个简单的序列二元分类器：

In [19]:
```
np.random.seed(42)
tf.random.set_seed(42)

embedding_size = 5
```

```
model = tf.keras.Sequential([
    tf.keras.layers.InputLayer(input_shape=[None],
                               dtype=tf.int32, ragged=True),
    tf.keras.layers.Embedding(input_dim=len(POSSIBLE_CHARS),
                              output_dim=embedding_size),
    tf.keras.layers.GRU(30),
    tf.keras.layers.Dense(1, activation="sigmoid")
])

optimizer = tf.keras.optimizers.SGD(learning_rate=0.02, momentum = 0.95,
                                    nesterov=True)

model.compile(loss="binary_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])

history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
```

```
Epoch 1/20
```

```
C:\Users\tu'tu\AppData\Roaming\Python\Python38\site-packages\tensorflow\python\framework\indexed_slices.py:444: UserWarning: Converting sparse IndexedSlices(IndexedSlices(indices=Tensor("gradient_tape/sequential/gru/RaggedToTensor/boolean_mask_1/GatherV2:0", shape=(None,), dtype=int32), values=Tensor("gradient_tape/sequential/gru/RaggedToTensor/boolean_mask/GatherV2:0", shape=(None, 5), dtype=float32), dense_shape=Tensor("gradient_tape/sequential/gru/RaggedToTensor/Shape:0", shape=(2,), dtype=int32))) to a dense Tensor of unknown shape. This may consume a large amount of memory.
  warnings.warn(
```

```
313/313 [==============================] - 8s 9ms/step - loss: 0.6888 - accuracy: 0.
5345 - val_loss: 0.6770 - val_accuracy: 0.4835
Epoch 2/20
313/313 [==============================] - 2s 8ms/step - loss: 0.6624 - accuracy: 0.
5672 - val_loss: 0.6578 - val_accuracy: 0.5075
Epoch 3/20
313/313 [==============================] - 2s 8ms/step - loss: 0.6465 - accuracy: 0.
5899 - val_loss: 0.6450 - val_accuracy: 0.5430
Epoch 4/20
313/313 [==============================] - 2s 8ms/step - loss: 0.6278 - accuracy: 0.
6039 - val_loss: 0.6284 - val_accuracy: 0.6320
Epoch 5/20
313/313 [==============================] - 2s 8ms/step - loss: 0.5784 - accuracy: 0.
6683 - val_loss: 0.5537 - val_accuracy: 0.6440
Epoch 6/20
313/313 [==============================] - 2s 8ms/step - loss: 0.4987 - accuracy: 0.
7459 - val_loss: 0.4340 - val_accuracy: 0.7990
Epoch 7/20
313/313 [==============================] - 2s 8ms/step - loss: 0.3344 - accuracy: 0.
8633 - val_loss: 0.2414 - val_accuracy: 0.9205
Epoch 8/20
313/313 [==============================] - 2s 8ms/step - loss: 0.1914 - accuracy: 0.
9349 - val_loss: 0.1063 - val_accuracy: 0.9680
Epoch 9/20
313/313 [==============================] - 2s 8ms/step - loss: 0.0833 - accuracy: 0.
9783 - val_loss: 0.0192 - val_accuracy: 0.9940
Epoch 10/20
313/313 [==============================] - 2s 8ms/step - loss: 0.0963 - accuracy: 0.
9732 - val_loss: 0.0105 - val_accuracy: 0.9990
Epoch 11/20
313/313 [==============================] - 2s 8ms/step - loss: 0.0056 - accuracy: 0.
9989 - val_loss: 0.0075 - val_accuracy: 0.9975
Epoch 12/20
313/313 [==============================] - 2s 7ms/step - loss: 0.0012 - accuracy: 1.
0000 - val_loss: 6.1697e-04 - val_accuracy: 1.0000
Epoch 13/20
313/313 [==============================] - 2s 8ms/step - loss: 5.5451e-04 - accurac
y: 1.0000 - val_loss: 4.3552e-04 - val_accuracy: 1.0000
Epoch 14/20
313/313 [==============================] - 3s 8ms/step - loss: 4.2101e-04 - accurac
y: 1.0000 - val_loss: 3.4947e-04 - val_accuracy: 1.0000
Epoch 15/20
313/313 [==============================] - 2s 8ms/step - loss: 3.4447e-04 - accurac
y: 1.0000 - val_loss: 2.9025e-04 - val_accuracy: 1.0000
Epoch 16/20
313/313 [==============================] - 2s 8ms/step - loss: 2.9172e-04 - accurac
y: 1.0000 - val_loss: 2.4781e-04 - val_accuracy: 1.0000
Epoch 17/20
313/313 [==============================] - 2s 8ms/step - loss: 2.5307e-04 - accurac
y: 1.0000 - val_loss: 2.1773e-04 - val_accuracy: 1.0000
Epoch 18/20
313/313 [==============================] - 2s 8ms/step - loss: 2.2357e-04 - accurac
y: 1.0000 - val_loss: 1.9310e-04 - val_accuracy: 1.0000
Epoch 19/20
313/313 [==============================] - 3s 8ms/step - loss: 1.9999e-04 - accurac
y: 1.0000 - val_loss: 1.7352e-04 - val_accuracy: 1.0000
```

```
Epoch 20/20
313/313 [==============================] - 2s 8ms/step - loss: 1.8106e-04 - accurac
y: 1.0000 - val_loss: 1.5805e-04 - val_accuracy: 1.0000
```

现在让我们在两个棘手的字符串上测试我们的 RNN：第一个是坏的而第二个是好的。它们仅在倒数第二个字符上有所不同。如果 RNN 做对了，则表明它设法注意到第二个字母应始终等于倒数第二个字母的模式。这需要相当长的短期记忆（这就是我们使用 GRU 单元的原因）。

In [20]:
```
test_strings = ["BPBTSSSSSSSXXTTVPXVPXTTTTTVVETE",
                "BPBTSSSSSSSXXTTVPXVPXTTTTTVVEPE"]
X_test = tf.ragged.constant([string_to_ids(s) for s in test_strings], ragged_rank=1

y_proba = model.predict(X_test)

print()

print("Estimated probability that these are Reber strings:")
for index, string in enumerate(test_strings):
    print("{}: {:.2f}%".format(string, 100 * y_proba[index][0]))
```

```
1/1 [==============================] - 0s 313ms/step

Estimated probability that these are Reber strings:
BPBTSSSSSSSXXTTVPXVPXTTTTTVVETE: 0.02%
BPBTSSSSSSSXXTTVPXVPXTTTTTVVEPE: 99.92%
```

哒哒！它运作良好。RNN 非常自信地找到了正确答案。

Exercise 9

训练一个编码器-解码器模型，它可以将日期字符串从一种格式转换为另一种格式（例如，从“April 22, 2019”到“2019-04-22”）。

答案：

让我们从创建数据集开始。我们将使用 1000-01-01 和 9999-12-31 之间的随机日期：

In [21]:
```
from datetime import date

# cannot use strftime()'s %B format since it depends on the locale
MONTHS = ["January", "February", "March", "April", "May", "June",
          "July", "August", "September", "October", "November", "December"]

def random_dates(n_dates):
    min_date = date(1000, 1, 1).toordinal()
    max_date = date(9999, 12, 31).toordinal()

    ordinals = np.random.randint(max_date - min_date, size=n_dates) + min_date
    dates = [date.fromordinal(ordinal) for ordinal in ordinals]

    x = [MONTHS[dt.month - 1] + " " + dt.strftime("%d, %Y") for dt in dates]
    y = [dt.isoformat() for dt in dates]
    return x, y
```

以下是一些随机日期，以输入格式和目标格式显示：

In [24]:
```
np.random.seed(42)

n_dates = 3
x_example, y_example = random_dates(n_dates)

print("{:25s}{:25s}".format("Input", "Target"))
print("-" * 50)

for idx in range(n_dates):
    print("{:25s}{:25s}".format(x_example[idx], y_example[idx]))
```

```
Input                     Target
--------------------------------------------------
September 20, 7075        7075-09-20
May 15, 8579              8579-05-15
January 11, 7103          7103-01-11
```

让我们获取输入中所有可能字符的列表：

In [25]:
```
INPUT_CHARS = "".join(sorted(set("".join(MONTHS) + "0123456789, ")))

INPUT_CHARS
```

Out[25]:
```
' ,0123456789ADFJMNOSabceghilmnoprstuvy'
```

这是输出中可能的字符列表：

In [27]:
```
OUTPUT_CHARS = "0123456789-"
```

让我们编写一个函数将字符串转换为字符 ID 列表，就像我们在上一个练习中所做的那样：

In [28]:
```
def date_str_to_ids(date_str, chars=INPUT_CHARS):
    return [chars.index(c) for c in date_str]
```

In [29]:
```
date_str_to_ids(x_example[0], INPUT_CHARS)
```

Out[29]:
```
[19, 23, 31, 34, 23, 28, 21, 23, 32, 0, 4, 2, 1, 0, 9, 2, 9, 7]
```

In [30]:
```
date_str_to_ids(y_example[0], OUTPUT_CHARS)
```

Out[30]:
```
[7, 0, 7, 5, 10, 0, 9, 10, 2, 0]
```

In [31]:
```
def prepare_date_strs(date_strs, chars=INPUT_CHARS):
    X_ids = [date_str_to_ids(dt, chars) for dt in date_strs]
    X = tf.ragged.constant(X_ids, ragged_rank=1)
    return (X + 1).to_tensor() # using 0 as the padding token ID

def create_dataset(n_dates):
    x, y = random_dates(n_dates)
    return prepare_date_strs(x, INPUT_CHARS), prepare_date_strs(y, OUTPUT_CHARS)
```

In [32]:
```
np.random.seed(42)

X_train, Y_train = create_dataset(10000)
X_valid, Y_valid = create_dataset(2000)
X_test, Y_test = create_dataset(2000)
```

In [40]:
```
X_train[1]
```

Out[40]:
```
<tf.Tensor: shape=(18,), dtype=int32, numpy=
array([17, 21, 38,  1,  4,  8,  2,  1, 11,  8, 10, 12,  0,  0,  0,  0,  0,
       0])>
```

First version: a very basic seq2seq model

让我们首先尝试最简单的模型：我们输入输入序列，它首先通过编码器（一个嵌入层，然后是一个 LSTM 层），输出一个向量，然后它通过一个解码器（一个 LSTM 层，后跟一个密集的输出层），它输出一系列向量，每个向量代表所有可能输出字符的估计概率。

由于解码器需要一个序列作为输入，因此我们将向量（由编码器输出）重复尽可能多的次数作为最长可能的输出序列。

In [41]:
```
embedding_size = 32
max_output_length = Y_train.shape[1]

np.random.seed(42)
tf.random.set_seed(42)

encoder = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=len(INPUT_CHARS) + 1,
                              output_dim=embedding_size,
                              input_shape=[None]),
    tf.keras.layers.LSTM(128)
])

decoder = tf.keras.Sequential([
    tf.keras.layers.LSTM(128, return_sequences=True),
    tf.keras.layers.Dense(len(OUTPUT_CHARS) + 1, activation="softmax")
])

model = tf.keras.Sequential([
    encoder,
    tf.keras.layers.RepeatVector(max_output_length),
    decoder
])

optimizer = tf.keras.optimizers.Nadam()

model.compile(loss="sparse_categorical_crossentropy",
              optimizer=optimizer, metrics=["accuracy"])

history = model.fit(X_train, Y_train, epochs=20,
                    validation_data=(X_valid, Y_valid))
```

```
Epoch 1/20
313/313 [==============================] - 9s 17ms/step - loss: 1.7985 - accuracy:
0.3556 - val_loss: 1.3465 - val_accuracy: 0.5137
Epoch 2/20
313/313 [==============================] - 4s 14ms/step - loss: 1.2800 - accuracy:
0.5408 - val_loss: 1.1351 - val_accuracy: 0.5861
Epoch 3/20
313/313 [==============================] - 4s 14ms/step - loss: 1.0331 - accuracy:
0.6339 - val_loss: 1.2092 - val_accuracy: 0.5715
Epoch 4/20
313/313 [==============================] - 4s 14ms/step - loss: 0.8746 - accuracy:
0.6824 - val_loss: 0.7306 - val_accuracy: 0.7201
Epoch 5/20
313/313 [==============================] - 4s 13ms/step - loss: 0.6270 - accuracy:
0.7601 - val_loss: 0.5399 - val_accuracy: 0.7901
Epoch 6/20
313/313 [==============================] - 4s 13ms/step - loss: 0.4413 - accuracy:
0.8285 - val_loss: 0.3635 - val_accuracy: 0.8591
Epoch 7/20
313/313 [==============================] - 4s 14ms/step - loss: 0.8324 - accuracy:
0.7045 - val_loss: 2.1387 - val_accuracy: 0.2990
Epoch 8/20
313/313 [==============================] - 4s 13ms/step - loss: 0.4764 - accuracy:
0.8273 - val_loss: 0.2845 - val_accuracy: 0.8938
Epoch 9/20
313/313 [==============================] - 4s 13ms/step - loss: 0.3571 - accuracy:
0.8788 - val_loss: 0.2056 - val_accuracy: 0.9327
Epoch 10/20
313/313 [==============================] - 4s 13ms/step - loss: 0.1683 - accuracy:
0.9492 - val_loss: 0.2000 - val_accuracy: 0.9351
Epoch 11/20
313/313 [==============================] - 4s 13ms/step - loss: 0.0991 - accuracy:
0.9760 - val_loss: 0.0734 - val_accuracy: 0.9851
Epoch 12/20
313/313 [==============================] - 4s 13ms/step - loss: 0.0550 - accuracy:
0.9913 - val_loss: 0.0449 - val_accuracy: 0.9931
Epoch 13/20
313/313 [==============================] - 4s 13ms/step - loss: 0.0326 - accuracy:
0.9968 - val_loss: 0.0286 - val_accuracy: 0.9966
Epoch 14/20
313/313 [==============================] - 4s 13ms/step - loss: 0.0201 - accuracy:
0.9988 - val_loss: 0.0180 - val_accuracy: 0.9985
Epoch 15/20
313/313 [==============================] - 4s 13ms/step - loss: 0.0975 - accuracy:
0.9777 - val_loss: 0.0605 - val_accuracy: 0.9898
Epoch 16/20
313/313 [==============================] - 4s 13ms/step - loss: 0.0223 - accuracy:
0.9983 - val_loss: 0.0151 - val_accuracy: 0.9987
Epoch 17/20
313/313 [==============================] - 4s 13ms/step - loss: 0.0092 - accuracy:
0.9999 - val_loss: 0.0088 - val_accuracy: 0.9995
Epoch 18/20
313/313 [==============================] - 4s 13ms/step - loss: 0.0060 - accuracy:
1.0000 - val_loss: 0.0063 - val_accuracy: 0.9998
Epoch 19/20
313/313 [==============================] - 4s 13ms/step - loss: 0.0043 - accuracy:
```

```
1.0000 - val_loss: 0.0048 - val_accuracy: 0.9999
Epoch 20/20
313/313 [==============================] - 4s 13ms/step - loss: 0.0033 - accuracy:
1.0000 - val_loss: 0.0038 - val_accuracy: 0.9999
```

看起来不错，我们达到了 100% 的验证准确率！让我们使用该模型进行一些预测。我们需要能够将字符 ID 序列转换为可读字符串：

In [43]:
```
def ids_to_date_strs(ids, chars=OUTPUT_CHARS):
    return ["".join([("?" + chars)[index] for index in sequence])
            for sequence in ids]
```

现在我们可以使用模型来转换一些日期：

In [44]:
```
X_new = prepare_date_strs(["September 17, 2009", "July 14, 1789"])
```

In [45]:
```
ids = model.predict(X_new).argmax(axis=-1)
for date_str in ids_to_date_strs(ids):
    print(date_str)
```

```
1/1 [==============================] - 1s 624ms/step
2009-09-17
1789-07-14
```

然而，由于该模型仅在长度为 18（最长日期的长度）的输入字符串上进行训练，因此如果我们尝试使用它对较短的序列进行预测，它的表现并不好：

In [46]:
```
X_new = prepare_date_strs(["May 02, 2020", "July 14, 1789"])
```

In [47]:
```
ids = model.predict(X_new).argmax(axis=-1)
for date_str in ids_to_date_strs(ids):
    print(date_str)
```

```
1/1 [==============================] - 1s 863ms/step
2020-01-02
1789-12-14
```

哎呀！我们需要确保我们始终传递与训练期间相同长度的序列，必要时使用填充。 让我们为此编写一个小辅助函数：

In [48]:
```
max_input_length = X_train.shape[1]

def prepare_date_strs_padded(date_strs):
    X = prepare_date_strs(date_strs)
    if X.shape[1] < max_input_length:
        X = tf.pad(X, [[0, 0], [0, max_input_length - X.shape[1]]])
    return X

def convert_date_strs(date_strs):
    X = prepare_date_strs_padded(date_strs)
    ids = model.predict(X).argmax(axis=-1)
    return ids_to_date_strs(ids)
```

In [49]:
```
convert_date_strs(["May 02, 2020", "July 14, 1789"])
```

```
1/1 [==============================] - 0s 31ms/step
```

Out[49]:
```
['2020-05-02', '1789-07-14']
```

诚然，编写日期转换工具肯定有更简单的方法（例如，使用正则表达式甚至基本的字符串操作），但您必须承认使用神经网络更酷。

然而，现实生活中的序列到序列问题通常会更难，所以为了完整起见，让我们构建一个更强大的模型。

Second version: feeding the shifted targets to the decoder (teacher forcing)

我们可以向解码器提供向右移动一个 time step 的目标序列，而不是向解码器提供编码器输出向量的简单重复。这样，在每个 time step，解码器都会知道前一个目标字符是什么。这应该有助于解决更复杂的序列到序列问题。

由于每个目标序列的第一个输出字符没有前一个字符，我们将需要一个新的标记来表示序列开始 (sos)。

在推理过程中，我们不知道目标，那么我们将向解码器提供什么？我们可以一次只预测一个字符，从一个 sos 标记开始，然后将到目前为止预测的所有字符提供给解码器（我们将在本笔记本的后面部分详细介绍）。

但是如果解码器的 LSTM 希望在每一步都得到之前的目标作为输入，我们应该如何将编码器输出的向量传给它呢？好吧，一种选择是忽略输出向量，而是使用编码器的 LSTM 状态作为解码器 LSTM 的初始状态（这要求编码器的 LSTM 必须与解码器的 LSTM 具有相同数量的单元）。

现在让我们创建解码器的输入（用于训练、验证和测试）。sos 字符将使用最后一个可能的输出字符的 ID + 1 来表示。

In [54]:
```
sos_id = len(OUTPUT_CHARS) + 1

def shifted_output_sequences(Y):
    sos_tokens = tf.fill(dims=(len(Y), 1), value=sos_id)
    return tf.concat([sos_tokens, Y[:, :-1]], axis=1)

X_train_decoder = shifted_output_sequences(Y_train)
X_valid_decoder = shifted_output_sequences(Y_valid)
X_test_decoder = shifted_output_sequences(Y_test)
```

让我们看一下解码器的训练输入：

In [56]:
```
X_train_decoder
```

Out[56]:
```
<tf.Tensor: shape=(10000, 10), dtype=int32, numpy=
array([[12,  8,  1, ..., 10, 11,  3],
       [12,  9,  6, ...,  6, 11,  2],
       [12,  8,  2, ...,  2, 11,  2],
       ...,
       [12, 10,  8, ...,  2, 11,  4],
       [12,  2,  2, ...,  3, 11,  3],
       [12,  8,  9, ...,  8, 11,  3]])>
```

现在让我们构建模型。它不再是一个简单的顺序模型，所以让我们使用函数式 API：

In [57]:
```
encoder_embedding_size = 32
decoder_embedding_size = 32
lstm_units = 128

np.random.seed(42)
tf.random.set_seed(42)

encoder_input = tf.keras.layers.Input(shape=[None], dtype=tf.int32)

encoder_embedding = tf.keras.layers.Embedding(
    input_dim=len(INPUT_CHARS) + 1,
    output_dim=encoder_embedding_size)(encoder_input)

_, encoder_state_h, encoder_state_c = tf.keras.layers.LSTM(
    lstm_units, return_state=True)(encoder_embedding)

encoder_state = [encoder_state_h, encoder_state_c]

decoder_input = tf.keras.layers.Input(shape=[None], dtype=tf.int32)

decoder_embedding = tf.keras.layers.Embedding(
    input_dim=len(OUTPUT_CHARS) + 2,
    output_dim=decoder_embedding_size)(decoder_input)

decoder_lstm_output = tf.keras.layers.LSTM(lstm_units, return_sequences=True)(
    decoder_embedding, initial_state=encoder_state)

decoder_output = tf.keras.layers.Dense(len(OUTPUT_CHARS) + 1,
                                       activation="softmax")(decoder_lstm_output)

model = tf.keras.Model(inputs=[encoder_input, decoder_input],
                       outputs=[decoder_output])

optimizer = tf.keras.optimizers.Nadam()

model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])

history = model.fit([X_train, X_train_decoder], Y_train, epochs=10,
                    validation_data=([X_valid, X_valid_decoder], Y_valid))
```

```
Epoch 1/10
313/313 [==============================] - 9s 17ms/step - loss: 1.6822 - accuracy:
0.3648 - val_loss: 1.4422 - val_accuracy: 0.4331
Epoch 2/10
313/313 [==============================] - 4s 14ms/step - loss: 1.2773 - accuracy:
0.5187 - val_loss: 1.0669 - val_accuracy: 0.5968
Epoch 3/10
313/313 [==============================] - 4s 14ms/step - loss: 0.7611 - accuracy:
0.7257 - val_loss: 0.4865 - val_accuracy: 0.8346
Epoch 4/10
313/313 [==============================] - 4s 14ms/step - loss: 0.2787 - accuracy:
0.9225 - val_loss: 0.1326 - val_accuracy: 0.9775
Epoch 5/10
313/313 [==============================] - 4s 14ms/step - loss: 0.0969 - accuracy:
0.9869 - val_loss: 0.0461 - val_accuracy: 0.9988
Epoch 6/10
313/313 [==============================] - 4s 14ms/step - loss: 0.0552 - accuracy:
0.9932 - val_loss: 0.0313 - val_accuracy: 0.9992
Epoch 7/10
313/313 [==============================] - 4s 14ms/step - loss: 0.0193 - accuracy:
0.9999 - val_loss: 0.0144 - val_accuracy: 0.9999
Epoch 8/10
313/313 [==============================] - 4s 14ms/step - loss: 0.0112 - accuracy:
1.0000 - val_loss: 0.0094 - val_accuracy: 0.9999
Epoch 9/10
313/313 [==============================] - 4s 14ms/step - loss: 0.0074 - accuracy:
1.0000 - val_loss: 0.0067 - val_accuracy: 1.0000
Epoch 10/10
313/313 [==============================] - 4s 14ms/step - loss: 0.0052 - accuracy:
1.0000 - val_loss: 0.0048 - val_accuracy: 1.0000
```

In [58]:
```python
sos_id = len(OUTPUT_CHARS) + 1

def predict_date_strs(date_strs):
    X = prepare_date_strs_padded(date_strs)
    Y_pred = tf.fill(dims=(len(X), 1), value=sos_id)
    for index in range(max_output_length):
        pad_size = max_output_length - Y_pred.shape[1]
        X_decoder = tf.pad(Y_pred, [[0, 0], [0, pad_size]])
        Y_probas_next = model.predict([X, X_decoder])[:, index:index+1]
        Y_pred_next = tf.argmax(Y_probas_next, axis=-1, output_type=tf.int32)
        Y_pred = tf.concat([Y_pred, Y_pred_next], axis=1)
    return ids_to_date_strs(Y_pred[:, 1:])
```

In [59]:
```python
predict_date_strs(["July 14, 1789", "May 01, 2020"])
```

```
1/1 [==============================] - 1s 599ms/step
1/1 [==============================] - 0s 29ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 29ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 27ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 27ms/step
1/1 [==============================] - 0s 28ms/step
```

Out[59]: ['1789-07-14', '2020-05-01']

Exercise 10

查看 Keras 网站上的"Natural language image search with a Dual Encoder"的示例。您将学习如何构建能够在同一嵌入空间中表示图像和文本的模型。这使得使用文本提示搜索图像成为可能，就像在 OpenAI 的 CLIP 模型中一样。

Exercise 11

使用 Hugging Face Transformer 库下载一个能够生成文本的预训练语言模型（例如，GPT），并尝试生成更有说服力的莎士比亚文本。您将需要使用模型的 generate() 方法——更多细节请参见 Hugging Face 的文档。

答案：

首先，让我们加载一个预训练模型。在这个例子中，我们将使用 OpenAI 的 GPT 模型，在顶部有一个额外的语言模型（只是一个线性层，其权重与输入嵌入相关联）。让我们导入它并加载预训练的权重（这会将大约 445MB 的数据下载到 ~/.cache/torch/transformers）：

In [61]:
```
from transformers import TFOpenAIGPTLMHeadModel

model = TFOpenAIGPTLMHeadModel.from_pretrained("openai-gpt")
```

```
Downloading (…)lve/main/config.json:   0%|          | 0.00/656 [00:00<?, ?B/s]
C:\ProgramData\Anaconda3\lib\site-packages\huggingface_hub\file_download.py:133: UserWarning: `huggingface_hub` cache-system uses symlinks by default to efficiently store duplicated files but your machine does not support them in C:\Users\tu'tu\.cache\huggingface\hub. Caching files will still work but in a degraded version that might require more space on your disk. This warning can be disabled by setting the `HF_HUB_DISABLE_SYMLINKS_WARNING` environment variable. For more details, see https://huggingface.co/docs/huggingface_hub/how-to-cache#limitations.
To support symlinks on Windows, you either need to activate Developer Mode or to run Python as an administrator. In order to see activate developer mode, see this article: https://docs.microsoft.com/en-us/windows/apps/get-started/enable-your-device-for-development
  warnings.warn(message)
Downloading tf_model.h5:   0%|          | 0.00/466M [00:00<?, ?B/s]
All model checkpoint layers were used when initializing TFOpenAIGPTLMHeadModel.

All the layers of TFOpenAIGPTLMHeadModel were initialized from the model checkpoint at openai-gpt.
If your task is similar to the task the model of the checkpoint was trained on, you can already use TFOpenAIGPTLMHeadModel for predictions without further training.
Downloading (…)neration_config.json:   0%|          | 0.00/74.0 [00:00<?, ?B/s]
```

接下来，我们将需要一个专门用于此模型的分词器。这个将尝试使用 spaCy 和 ftfy 库（如果已安装），否则它将退回到 BERT 的 BasicTokenizer，然后是字节对编码（对于大多数用例来说应该没问题）。

In [63]:
```
from transformers import OpenAIGPTTokenizer

tokenizer = OpenAIGPTTokenizer.from_pretrained("openai-gpt")
```

```
Downloading (…)olve/main/vocab.json:   0%|          | 0.00/816k [00:00<?, ?B/s]
Downloading (…)olve/main/merges.txt:   0%|          | 0.00/458k [00:00<?, ?B/s]
ftfy or spacy is not installed using BERT BasicTokenizer instead of SpaCy & ftfy.
```

现在让我们使用分词器对提示文本进行分词和编码：

In [64]:
```
tokenizer("hello everyone")
```

Out[64]:
```
{'input_ids': [3570, 1473], 'attention_mask': [1, 1]}
```

In [65]:
```
prompt_text = "This royal throne of kings, this sceptred isle"

encoded_prompt = tokenizer.encode(prompt_text,
                                  add_special_tokens=False,
                                  return_tensors="tf")
encoded_prompt
```

Out[65]:
```
<tf.Tensor: shape=(1, 10), dtype=int32, numpy=
array([[  616,  5751,  6404,   498,  9606,   240,   616, 26271,  7428,
        16187]])>
```

简单的！接下来，让我们使用模型在提示后生成文本。我们将生成 5 个不同的句子，每个句子都以提示文本开头，然后是 40 个额外的标记。要了解所有超参数的作用，请务必查看 Patrick von Platen（来自 Hugging Face）的这篇精彩博客文章。您可以尝试使用超参数来尝试获得更好的结果。

In [66]:
```
num_sequences = 5
length = 40

generated_sequences = model.generate(
    input_ids=encoded_prompt,
    do_sample=True,
    max_length=length + len(encoded_prompt[0]),
    temperature=1.0,
    top_k=0,
    top_p=0.9,
    repetition_penalty=1.0,
    num_return_sequences=num_sequences,
)

generated_sequences
```

Out[66]:
```
<tf.Tensor: shape=(5, 50), dtype=int32, numpy=
array([[  616,  5751,  6404,   498,  9606,   240,   616, 26271,  7428,
        16187, 40477,   544,   246, 15147,   562,   481,  9606,   498,
          481,  2868,   239, 40477,   620,   481,  5908,   498,   481,
         2868,   240,   556,   531,  2892,   945,   488,   524,   929,
         2784,   240,   498,  6228, 17379,   240, 40477,  9447,   485,
          524,  5353,  7339,   556,   524],
       [  616,  5751,  6404,   498,  9606,   240,   616, 26271,  7428,
        16187,   246,  1436,   239,   606,  1683,   793,   504,   246,
         6404,   498,  9753, 14386,   239,   606,  1259,  1683,   557,
        11907,   498,   616,   989,   239,   244, 40477,   664,   566,
          558,  7380, 28252,   481,   618,   240,   488,   664,   566,
         7380,  2071,   551,   498,  1300],
       [  616,  5751,  6404,   498,  9606,   240,   616, 26271,  7428,
        16187,   267, 40477,  1598,   481,  4187,   498,  1504,   260,
        34885,   535,  4761,   843,  1578,   481,  1820,   498,   616,
          260,  3722,   260,  1454,  2992,  2034,   524,   956,   240,
         1598,   512,   580,   246,  1092,   498,  1820,   500,   481,
         1424,  2525,   488,  2525,   498],
       [  616,  5751,  6404,   498,  9606,   240,   616, 26271,  7428,
        16187,   239, 40477,   500,   524,  2650,   240,   481,   618,
          509,  1886,   481,  5153,  1546,   498,  1114,   743,   585,
        28281,   240,   246,  1203,   260,  2286,  4198,   240,  1234,
          487,   509,   498,   481,  5751,  1584,   239,  2034,   524,
         1767,   240,   487,   636,  1362],
       [  616,  5751,  6404,   498,  9606,   240,   616, 26271,  7428,
        16187,   498,  2935,  1137,  1043,   524,  6404,   240,   260,
          246,  1424,  2264,   498,   481,  9606,   498,   653,   496,
        17265,   488,   498,   481,   618,  1742,  3446,   525,   980,
          694,  3344,   793,  2932,   240,   568,   871,   525,  1657,
         1896,   812,   580, 11505,   240]])>
```

现在让我们解码生成的序列并打印它们：

In [67]:
```
for sequence in generated_sequences:
    text = tokenizer.decode(sequence, clean_up_tokenization_spaces=True)
    print(text)
    print("-" * 80)
```

```
this royal throne of kings, this sceptred isle
 is a beacon for the kings of the ages.
 so the hero of the ages, with an older son and his first mate, of stolen runes,
 returns to his former kingdom with his
--------------------------------------------------------------------------------
this royal throne of kings, this sceptred isle a god. we sit here on a throne of ble
ssed harmony. we must sit as ruler of this people. "
 no one had dared contradict the king, and no one dared speak out of turn
--------------------------------------------------------------------------------
this royal throne of kings, this sceptred isle!
 may the gods of ah - puch's balance still hold the power of this - tent - white fle
sh upon his body, may you be a light of power in the great heat and heat of
--------------------------------------------------------------------------------
this royal throne of kings, this sceptred isle.
 in his choice, the king was given the kingship of house gidedrian, a three - headed
dragon, since he was of the royal line. upon his death, he would name
--------------------------------------------------------------------------------
this royal throne of kings, this sceptred isle of silver stood behind his throne, -
a great hall of the kings of earendil and of the king baldor that has been hidden he
re forever, but ever that whole seat will be vacant,
--------------------------------------------------------------------------------
```

您可以尝试更新的（和更大的）模型，例如 GPT-2、CTRL、Transformer-XL 或 XLNet ，它们都可以作为 transformers 库中的预训练模型使用，包括顶部带有语言模型的变体。模型之间的预处理步骤略有不同，因此请务必查看 transformer 文档中的这个生成示例（此示例使用 PyTorch，但只需很少的调整即可工作，例如在模型类名称的开头添加 `TF`，删除 `.to()` 方法调用，并使用 `return_tensors="tf"` 而不是 `"pt"`。