NLP_lecture

November 15, 2020

Chapter 16 – Natural Language Processing with RNNs and Attention

This notebook contains all the sample code in chapter 16.

Run in Google Colab

1 Setup

First, let's import a few common modules, ensure MatplotLib plots figures inline and prepare a function to save the figures. We also check that Python 3.5 or later is installed (although Python 2.x may work, it is deprecated so we strongly recommend you use Python 3 instead), as well as Scikit-Learn ≥0.20 and TensorFlow ≥2.0.

[2]:
```
# Python ≥3.5 is required
import sys
assert sys.version_info >= (3, 5)

# Scikit-Learn ≥0.20 is required
import sklearn
assert sklearn.__version__ >= "0.20"

try:
    # %tensorflow_version only exists in Colab.
    %tensorflow_version 2.x
    !pip install -q -U tensorflow-addons
    IS_COLAB = True
except Exception:
    IS_COLAB = False

# TensorFlow ≥2.0 is required
import tensorflow as tf
from tensorflow import keras
assert tf.__version__ >= "2.0"

if not tf.config.list_physical_devices('GPU'):
    print("No GPU was detected. LSTMs and CNNs can be very slow without a GPU.")
    if IS_COLAB:
        print("Go to Runtime > Change runtime and select a GPU hardware␣
↪accelerator.")
```

```
# Common imports
import numpy as np
import os

# to make this notebook's output stable across runs
np.random.seed(42)
tf.random.set_seed(42)

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)

# Where to save the figures
PROJECT_ROOT_DIR = "."
CHAPTER_ID = "nlp"
IMAGES_PATH = os.path.join(PROJECT_ROOT_DIR, "images", CHAPTER_ID)
os.makedirs(IMAGES_PATH, exist_ok=True)

def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=300):
    path = os.path.join(IMAGES_PATH, fig_id + "." + fig_extension)
    print("Saving figure", fig_id)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
```

```
No GPU was detected. LSTMs and CNNs can be very slow without a GPU.
```

NLP is a classcial goal of computer science starting with Alan Turing. Humans can be sometimes fooled by charbots with hard-coded rules: “If human says”How are you?” then answer “I am fine”. But mastering a language is very difficult task. Another important reason to study NLP is that most of the human knowledge is presented in form of text and it reqires mastering of a language.

A common method for NLP is RNN, because a test/dialog is a series of letters, words or sentences connected to each other. After the RNN model is trained on a corpus (large collection of texts) it can be used to generate new texts, or at least predict the next charater/word/sentence.

First we will cover the *stateless* RNN which treats each batch of text separately, like we treated a random draw of 20 days of stock market data. This model does not connect a sample with the rest of the series/text. Next we will study *stateful* RNN, which runs through the whole data preserving the hidden states of RNN as it moves from one batch to the next. We will use this to predict what would the Shakespeare write next.

Then we will use RNNs for sentimental analysis of moview reviews (rater’s emotional attituted to the movie). Then we will use Encoder–Decoder architecture capable of performingmneural

machine translation (NMT). Then we look at a very successful attentiononly architecture called the Transformer. Finally, we will take a look at some of the most important advances in NLP in the recent years, including incredibly powerful language models such as GPT-2, GPT-3, and BERT, both based on Transformers.

2 Char-RNN

Char-RNN can then be used to generate novel text, one character at a time. We will train it on the full corpus of Shakespeare. Example of Shakespeare like text generated by a model:

Alas, I think he shall be come approached and the day When little srain would be attain'd into being never fed, And who is but a chain and subjects of his death, I should not sleep.

2.1 Splitting a sequence into batches of shuffled windows

For example, let's split the sequence 0 to 14 into windows of length 5, each shifted by 2 (e.g.,`[0, 1, 2, 3, 4]`, `[2, 3, 4, 5, 6]`, etc.), then shuffle them, and split them into inputs (the first 4 steps) and targets (the last 4 steps) (e.g., `[2, 3, 4, 5, 6]` would be split into `[[2, 3, 4, 5], [3, 4, 5, 6]]`), then create batches of 3 such input/target pairs:

[3]:
```
np.random.seed(42)
tf.random.set_seed(42)

n_steps = 5
# get a list from 0 to 14
dataset = tf.data.Dataset.from_tensor_slices(tf.range(15))
dataset = dataset.window(n_steps, shift=2, drop_remainder=True)
# divide data into batches with 5 steps (periods), the batches should start␣
 ↪from every other digit [2,4,6]
dataset = dataset.flat_map(lambda window: window.batch(n_steps))
# flatten the vector
# shuffle(10) loads 10 observations in the memory and shufles them, and then␣
 ↪adds another 10. This parameters
# is added to reduce the memory requirements for very large dataset. Here it is␣
 ↪just for illustration.
dataset = dataset.shuffle(10).map(lambda window: (window[:-1], window[1:]))
# get batches with 3 observations in each. # Then the whole dataset will be␣
 ↪covered in 2 batches.
#Prefetch just prepares one batch as we build the one. It improves speed,
# and have not effect on results. You can try different values with prefetch␣
 ↪and you will get identical retults.
dataset = dataset.batch(3).prefetch(1)
# get
for index, (X_batch, Y_batch) in enumerate(dataset):
    print("_" * 20, "Batch", index, "\nX_batch")
    print(X_batch.numpy())
    print("=" * 5, "\nY_batch")
    print(Y_batch.numpy())
```

```
____________________ Batch 0
X_batch
[[6 7 8 9]
 [2 3 4 5]
 [4 5 6 7]]
=====
Y_batch
[[ 7  8  9 10]
 [ 3  4  5  6]
 [ 5  6  7  8]]
____________________ Batch 1
X_batch
[[ 0  1  2  3]
 [ 8  9 10 11]
 [10 11 12 13]]
=====
Y_batch
[[ 1  2  3  4]
 [ 9 10 11 12]
 [11 12 13 14]]
```

2.2 Loading the Data and Preparing the Dataset

[3]:
```
#Load Shakespeare
shakespeare_url = "https://raw.githubusercontent.com/karpathy/char-rnn/master/
 ↪data/tinyshakespeare/input.txt"
filepath = keras.utils.get_file("shakespeare.txt", shakespeare_url)
with open(filepath) as f:
    shakespeare_text = f.read()
```

[4]:
```
print(shakespeare_text[:148])
# Hint: This is the start of the Tragedy of the Coriolanes
```

```
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You are all resolved rather to die than to famish?
```

Let's look at the set of all characters used in the text:

[5]:
```
char_list = "".join(sorted(set(shakespeare_text.lower())))
print(char_list)
print(f"The list has {len(char_list)} elements")
```

```
 !$&',-.3:;?abcdefghijklmnopqrstuvwxyz
The list has 39 elements
```

Next, we encode every character as an integer for prediction. We will use Keras's Tokenizer class that will do the conversion for us.

The tokenizer will find all the characters used in the text and map each of them to a different character ID, from 1 to the number of distinct characters (it starts from 1, rather than from 0).

[6]:
```
# load tokenizer
tokenizer = keras.preprocessing.text.Tokenizer(char_level=True)
# Fit tokenizer for our corpus:
tokenizer.fit_on_texts(shakespeare_text)
```

[7]:
```
tokenizer.texts_to_sequences(["First"])
# codes for the letters
```

[7]:
```
[[20, 6, 9, 8, 3]]
```

[8]:
```
tokenizer.sequences_to_texts([[20, 6, 9, 8, 3]])
#going back we will get first. By default the tokenizer will make all␣
 ↪characters low case, if you want to keep upper
# case charachters separate use:  = keras.preprocessing.text.
 ↪Tokenizer(char_level=True, lower=False)
```

[8]:
```
['f i r s t']
```

[9]:
```
max_id = len(tokenizer.word_index) # number of distinct characters
dataset_size = tokenizer.document_count # total number of characters
print(max_id, dataset_size)
# 39 distinct characters, and about 1 million characters
```

```
39 1115394
```

[10]:
```
# Encode characters as integerst to save space. In Python integer is 24 bytes,␣
 ↪string is 58 bytes.
print(sys.getsizeof((10)))
print(sys.getsizeof(('f')))
# Array starts from 0, so the token 1 becomes token 0.
[encoded] = np.array(tokenizer.texts_to_sequences([shakespeare_text])) - 1
# 'F' become 19, rather than 20 it was before
print([encoded][0:5])
# get training data, first 90% of the corpus
train_size = dataset_size * 90 // 100
dataset = tf.data.Dataset.from_tensor_slices(encoded[:train_size])
```

```
28
58
```

```
[array([19,  5,  8, …, 20, 26, 10])]
```

- We need to split the dataset into a training, validation, and test sets. We cannot shuffle all characters – the text will become meaningless. So, we need to select the chunks of texts. The sets will be need to separated, so the same sentences/paragraphs don't appear in different sets.
- Splitting the time-series is a difficult task: splitting 2012-2015 vs 2016-2018 will introduce bias, as would splitting Hamlet from Romeo and Juliet. The works may be structurally different. The trade off: how to preserve enough structure for training/testing without being biased from training/testing sets having different structure. The high quality split may take a lot of trials and errors.
- We will simply takes first 90% of the text for training and the rest of testing/validation.

Our training set has now 1 million characters. Training over it in one go would require NN with millions of neurons, which will take forever and may result in over-fitting. Instead we will use window() method to convert his long sequence of characters into many smaller windows of text.

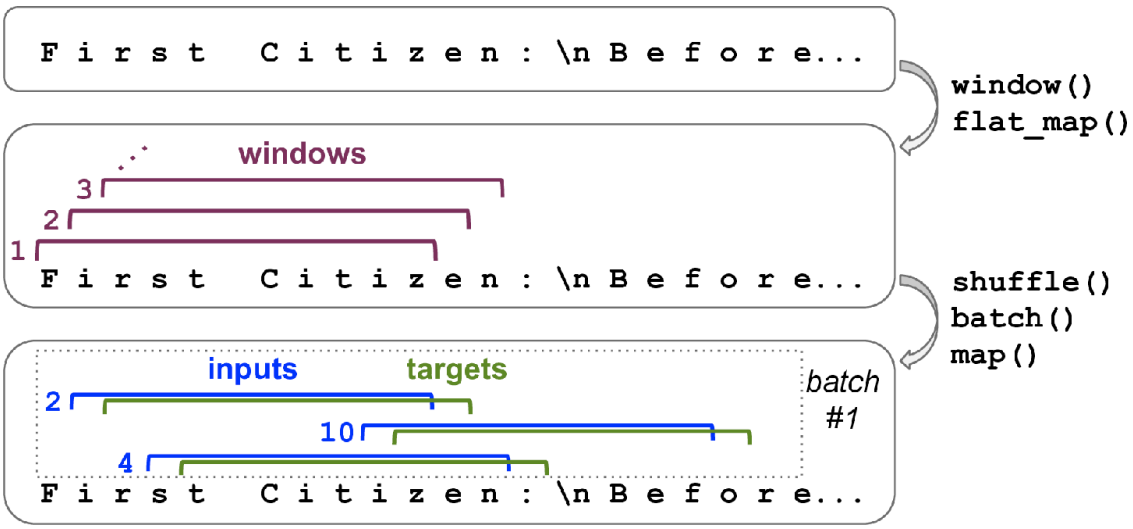
Every training instance will be a short substring of the whole text, and the RNN will be unrolled only over the length of these substrings. This is called truncated backpropagation through time.

[11]:
```
# Take a window of 101 characters: X will be 100 characters and 101st character␣
↪will be predicted by RNN.
n_steps = 100
window_length = n_steps + 1 # target = input shifted 1 character ahead
# Get windows from text shifting by 1, so we predict each of character out of␣
↪1M based on the 100 preceeding
#characters
dataset = dataset.repeat().window(window_length, shift=1, drop_remainder=True)
```

[12]:
```
# Flatten the vectors
dataset = dataset.flat_map(lambda window: window.batch(window_length))
```

[13]:
```
np.random.seed(42)
tf.random.set_seed(42)
```

[14]:
```
batch_size = 32
# Get baches of 32 phrases. Shuffle and break into batches 10K series at a time
dataset = dataset.shuffle(10000).batch(batch_size)
# break data into X and Y
dataset = dataset.map(lambda windows: (windows[:, :-1], windows[:, 1:]))
```



Categorical input features should be encoded as one-hot vectors or as embeddings. Here, we will encode each character using a one-hot vector because there are few distinct characters (only 39):

[15]:
```
dataset = dataset.map(
    # one-hot vector
    lambda X_batch, Y_batch: (tf.one_hot(X_batch, depth=max_id), Y_batch))
```

[16]:
```
# just add prefetchin for speed
dataset = dataset.prefetch(1)
```

[17]:
```
# show the first batch
for X_batch, Y_batch in dataset.take(1):
    print(X_batch[0], Y_batch[0], X_batch[0][0])
    print("Shapes")
    print(X_batch.shape, Y_batch.shape)
# categorical input features should generally be encoded, usually as one-hot␣
 ↪vectors or as embeddings.
#There are only 39, so we use one-hot vector.
```

```
tf.Tensor(
[[0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]
 …
 [0. 1. 0. … 0. 0. 0.]
 [1. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]], shape=(100, 39), dtype=float32) tf.Tensor(
[ 5  7  0  7  4 15  7  0  2  6  1  0 21  1 11 11 15 17  0 14  4  8 24  0
 14  1 17 31 31 10 10 19  5  8  7  2  0 18  5  2  5 35  1  9 23 10  4 15
 17  0  7  5  8 28  0 16  1 11 11 17  0 16  1 11 11 26 10 10 14  1  9  1
  9  5 13  7 23 10 27  2  6  3 13 20  6  0  4 11 11  0  4  2  0  3  9 18
  1  0 18  4], shape=(100,), dtype=int64) tf.Tensor(
```

```
[0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.], shape=(39,), dtype=float32)
Shapes
(32, 100, 39) (32, 100)
```

[ ]:

2.3 Creating and Training the Model

We predict the next character based on the previous 100 characters using RNN with 2 GRU layers of 128 units each and 20% dropout on both the inputs and hiddens states.

The output layer is a time-distributed Dense layer with 39 neurons.We apply the softmax activation function to pick most the character with the highest probability.

On my fast computer one epoch takes a bit over one hour, so the estimation takes about 10 hours.

[18]:
```
model = keras.models.Sequential([
    keras.layers.GRU(128, return_sequences=True, input_shape=[None, max_id],
                     dropout=0.2, recurrent_dropout=0.2),
    keras.layers.GRU(128, return_sequences=True,
                     dropout=0.2, recurrent_dropout=0.2),
    keras.layers.TimeDistributed(keras.layers.Dense(max_id,
                                                    activation="softmax"))
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam")
history = model.fit(dataset, steps_per_epoch=train_size // batch_size,
                    epochs=10)
```

```
Train for 31370 steps
Epoch 1/10
31370/31370 [==============================] - 7150s 228ms/step - loss: 1.4671
Epoch 2/10
31370/31370 [==============================] - 7094s 226ms/step - loss: 1.3614
Epoch 3/10
31370/31370 [==============================] - 7063s 225ms/step - loss: 1.3404
Epoch 4/10
31370/31370 [==============================] - 7039s 224ms/step - loss: 1.3311
Epoch 5/10
31370/31370 [==============================] - 7056s 225ms/step - loss: 1.3256
Epoch 6/10
31370/31370 [==============================] - 7049s 225ms/step - loss: 1.3209
Epoch 7/10
31370/31370 [==============================] - 7068s 225ms/step - loss: 1.3166
Epoch 8/10
31370/31370 [==============================] - 7030s 224ms/step - loss: 1.3138
Epoch 9/10
31370/31370 [==============================] - 7061s 225ms/step - loss: 1.3120
Epoch 10/10
```

```
31370/31370 [==============================] - 7177s 229ms/step - loss: 1.3105
```

[29]:
```
def preprocess(texts):
    X = np.array(tokenizer.texts_to_sequences(texts)) - 1
    return tf.one_hot(X, max_id)
```

[30]:
```
X_new = preprocess(["How are yo"])
Y_pred = model.predict_classes(X_new)
tokenizer.sequences_to_texts(Y_pred + 1)[0][-1] # 1st sentence, last char
```

```
WARNING:tensorflow:From <ipython-input-30-f85cbe487a4c>:2:
Sequential.predict_classes (from tensorflow.python.keras.engine.sequential) is
deprecated and will be removed after 2021-01-01.
Instructions for updating:
Please use instead:* `np.argmax(model.predict(x), axis=-1)`,   if your model
does multi-class classification   (e.g. if it uses a `softmax` last-layer
activation).* `(model.predict(x) > 0.5).astype("int32")`,   if your model does
binary classification   (e.g. if it uses a `sigmoid` last-layer activation).
```

[30]:
```
'u'
```

[31]:
```
tf.random.set_seed(42)

tf.random.categorical([[np.log(0.5), np.log(0.4), np.log(0.1)]],␣
↪num_samples=40).numpy()
```

[31]:
```
array([[0, 1, 0, 2, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 2, 1, 0, 2, 1,
        0, 1, 2, 1, 1, 1, 2, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 2]])
```

[32]:
```
def next_char(text, temperature=1):
    X_new = preprocess([text])
    y_proba = model.predict(X_new)[0, -1:, :]
    rescaled_logits = tf.math.log(y_proba) / temperature
    char_id = tf.random.categorical(rescaled_logits, num_samples=1) + 1
    return tokenizer.sequences_to_texts(char_id.numpy())[0]
```

[33]:
```
tf.random.set_seed(42)

next_char("How are yo", temperature=1)
```

[33]:
```
'u'
```

[34]:
```
def complete_text(text, n_chars=50, temperature=1):
    for _ in range(n_chars):
        text += next_char(text, temperature)
    return text
```

[35]:
```
tf.random.set_seed(42)

print(complete_text("t", temperature=0.2))
```

```
ting the country.

clown:
the soul in the bloody to
```

[36]:
```
print(complete_text("t", temperature=1))
```

```
thing! and yours?
who, thou art conpumber deeping h
```

[37]:
```
print(complete_text("t", temperature=2))
```

```
th no fwegrs hofje
of a brot. i bline if widoc rus:
```

[18]:
```
model = keras.models.Sequential([
    keras.layers.GRU(128, return_sequences=True, input_shape=[None, max_id],
                     dropout=0.2, recurrent_dropout=0.2),
    keras.layers.GRU(128, return_sequences=True,
                     dropout=0.2, recurrent_dropout=0.2),
    keras.layers.TimeDistributed(keras.layers.Dense(max_id,
                                                    activation="softmax"))
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam")
history = model.fit(dataset, steps_per_epoch=train_size // batch_size,
                    epochs=10)
```

```
Train for 31370 steps
Epoch 1/10
31370/31370 [==============================] - 7150s 228ms/step - loss: 1.4671
Epoch 2/10
31370/31370 [==============================] - 7094s 226ms/step - loss: 1.3614
Epoch 3/10
31370/31370 [==============================] - 7063s 225ms/step - loss: 1.3404
Epoch 4/10
31370/31370 [==============================] - 7039s 224ms/step - loss: 1.3311
Epoch 5/10
31370/31370 [==============================] - 7056s 225ms/step - loss: 1.3256
Epoch 6/10
31370/31370 [==============================] - 7049s 225ms/step - loss: 1.3209
Epoch 7/10
31370/31370 [==============================] - 7068s 225ms/step - loss: 1.3166
Epoch 8/10
31370/31370 [==============================] - 7030s 224ms/step - loss: 1.3138
```

```
Epoch 9/10
31370/31370 [==============================] - 7061s 225ms/step - loss: 1.3120
Epoch 10/10
31370/31370 [==============================] - 7177s 229ms/step - loss: 1.3105
```

2.4 Using the Model to Generate Text

[19]:
```
def preprocess(texts):
    X = np.array(tokenizer.texts_to_sequences(texts)) - 1
    return tf.one_hot(X, max_id)
```

[20]:
```
# convert sentence to vector
X_new = preprocess(["How are yo"])
# predict next character
Y_pred = model.predict_classes(X_new)
# show prediction
tokenizer.sequences_to_texts(Y_pred + 1)[0][-1] # 1st sentence, last char
```

[20]:
```
'u'
```

We could generate new text using the Char-RNN model by feeding it some text, make the model predict the most likely next letter, add it at the end of the text, then give the extended text to the model to guess the next letter, and so on.

In practice this often leads to the same words being repeated over and over again.

So, instead, we pick the next character randomly, with a probability equal to the estimated probability, using TensorFlow's tf.random.categorical() function. This will generate more diverse and interesting text.

The categorical() function samples random class indices, given the class log probabilities (logits).

For more control over the diversity of the generated text, we can divide the logits by a number called the temperature: a temperature close to 0 will favor the high-probability characters, while a very high temperature will give all characters an equal probability.

[21]:
```
#Example of the sample of logits
tf.random.set_seed(42)

tf.random.categorical([[np.log(0.5), np.log(0.4), np.log(0.1)]],␣
 ↪num_samples=40).numpy()
```

[21]:
```
array([[0, 1, 0, 2, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 2, 1, 0, 2, 1,
        0, 1, 2, 1, 1, 1, 2, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 2]])
```

[22]:
```
def next_char(text, temperature=1):
    X_new = preprocess([text])
    y_proba = model.predict(X_new)[0, -1:, :]
    rescaled_logits = tf.math.log(y_proba) / temperature
    char_id = tf.random.categorical(rescaled_logits, num_samples=1) + 1
```

```
    return tokenizer.sequences_to_texts(char_id.numpy())[0]
```

[23]:
```
tf.random.set_seed(42)

next_char("How are yo", temperature=1)
```

[23]:
```
'u'
```

[24]:
```
def complete_text(text, n_chars=50, temperature=1):
    for _ in range(n_chars):
        text += next_char(text, temperature)
    return text
```

[25]:
```
tf.random.set_seed(42)

print(complete_text("t", temperature=0.2))
```

```
WARNING:tensorflow:5 out of the last 6 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:6 out of the last 7 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:7 out of the last 8 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:8 out of the last 9 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
```

```
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:9 out of the last 10 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:10 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:10 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:10 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:10 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:10 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
```

```
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:10 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:10 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:10 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:10 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:10 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 12 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
```

```
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
```

```
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
```

```
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
```

```
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
```

```
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
WARNING:tensorflow:11 out of the last 11 calls to <function
_make_execution_function.<locals>.distributed_function at 0x7f8d44616830>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings is likely due to passing python objects instead of tensors. Also,
tf.function has experimental_relax_shapes=True option that relaxes argument
shapes that can avoid unnecessary retracing. Please refer to https://www.tensorf
```

```
low.org/tutorials/customization/performance#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for more details.
the belly the charges of the other words
and belly
```

[44]:
```
print(complete_text("t", temperature=0.5))
```

```
tion:
then is the state end my father so seems to t
```

[45]:
```
print(complete_text("Truth", temperature=0.7))
```

```
Truth,
how the time the sun gates i noble soul dardente
```

[39]:
```
tf.random.set_seed(42)
#You can see how temperature increases randomness
print(next_char("How are yo", temperature=1))
print(next_char("How are yo", temperature=3))
print(next_char("How are yo", temperature=5))
print(next_char("How are yo", temperature=10))
print(next_char("How are yo", temperature=20))
```

```
u
u
-
x
m
```

If you want to see state-of-the-art GPT-3 text generation check this out: https://arr.am/2020/07/14/elon-musk-by-dr-seuss-gpt-3/

Once there was a man who really was a Musk. He liked to build robots and rocket ships and such. He said, "I'm building a car that's electric and cool. I'll bet it outsells those Gasoline-burning clunkers soon!"

Apparently our Shakespeare model works best at a temperature close to 1. But the results are not great, what we can do:

1. To generate more convincing text, you could try using more GRU layers and more neurons per layer, train for longer, and add some regularization.
2. The major drawback of the model – it's incapable of learning patterns longer than 100 characters. We could make the window larger, but it will also make training harder, and even LSTM and GRU cells cannot handle very long sequences.
3. Alternatively, you could use a stateful RNN.

2.5 Stateful RNN

Until now, we have used only stateless RNNs: - each training iteration the model starts with a hidden state full of zeros, - then it updates this state at each time step, - after the last time step, it throws it away, as it is not needed anymore.

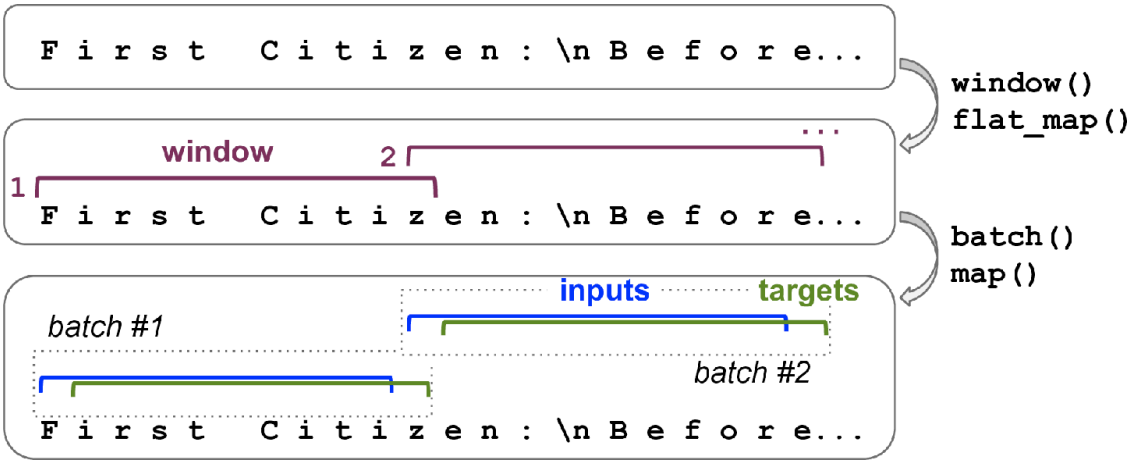
Stateful RNN can preserve its final state after processing one training batch and use it as the initial state for the next training batch. This way the model can learn long-term patterns despite only backpropagating through short sequences.

Stateful RNN needs each input sequence in a batch to start exactly where the corresponding sequence in the previous batch left off. We need to do to build a stateful RNN is to use sequential and nonoverlaping input sequences (rather than the shuffled and overlapping sequences we used to train stateless RNNs).

We will use shift=n_steps (instead of shift=1) when calling the window() method and would not use shuffle.

Batching is much harder with stateful RNN. batch(32) would produce 32 consecutive windows in the same the same batch, and the following batch would not continue each of these window where it left off.

The first batch would contain windows 1 to 32 and the second batch would contain windows 33 to 64, so if you consider, say, the first window of each batch (i.e., windows 1 and 33), you can see that they are not consecutive. The simplest solution to this problem is to just use "batches" containing a single window:



We could chop Shakespeare's text into 32 texts of equal length, create one dataset of consecutive input sequences for each of them, and finally use tf.train.Dataset.zip(datasets).map(lambda *windows: tf.stack(windows)) to create proper consecutive batches, where the nth input sequence in a batch starts off exactly where the nth input sequence ended in the previous batch (see the notebook for the full code).

[46]:
```
tf.random.set_seed(42)
```

[47]:
```
# get TF dataset from training data
dataset = tf.data.Dataset.from_tensor_slices(encoded[:train_size])
# break it by window with shift=n_steps (100)
dataset = dataset.window(window_length, shift=n_steps, drop_remainder=True)
# flatten the data
dataset = dataset.flat_map(lambda window: window.batch(window_length))
# get batches. repeat() the dataset resamples indefinitely.
dataset = dataset.repeat().batch(1)
# create overlapping windows function
dataset = dataset.map(lambda windows: (windows[:, :-1], windows[:, 1:]))
# get batches
dataset = dataset.map(
    lambda X_batch, Y_batch: (tf.one_hot(X_batch, depth=max_id), Y_batch))
dataset = dataset.prefetch(1)
```

[48]:
```
# set batch_size 32 consequitive windows
batch_size = 32
# split training data into 32 pieces
encoded_parts = np.array_split(encoded[:train_size], batch_size)
datasets = []
# for each of 32 batches:
for encoded_part in encoded_parts:
    # convert data to tensors
    dataset = tf.data.Dataset.from_tensor_slices(encoded_part)
    # get windows
    dataset = dataset.window(window_length, shift=n_steps, drop_remainder=True)
    # flatten the arrays
    dataset = dataset.flat_map(lambda window: window.batch(window_length))
    # append data
    datasets.append(dataset)
    # put the 32 as tuples back together
dataset = tf.data.Dataset.zip(tuple(datasets)).map(lambda *windows: tf.
 ↪stack(windows))
# attach the dataset
dataset = dataset.repeat().map(lambda windows: (windows[:, :-1], windows[:, 1:
 ↪]))
#get X and Y data for training
dataset = dataset.map(
    lambda X_batch, Y_batch: (tf.one_hot(X_batch, depth=max_id), Y_batch))
dataset = dataset.prefetch(1)
```

[49]:
```
model = keras.models.Sequential([
    # Pay attention to stateful = True
    keras.layers.GRU(128, return_sequences=True, stateful=True,
                     dropout=0.2, recurrent_dropout=0.2,
                     # RNN needs to know batch size to preserve the states in a␣
 ↪correct way
```

```
                    batch_input_shape=[batch_size, None, max_id]),
    keras.layers.GRU(128, return_sequences=True, stateful=True,
                     dropout=0.2, recurrent_dropout=0.2),
    keras.layers.TimeDistributed(keras.layers.Dense(max_id,
                                                    activation="softmax"))
])
```

[50]:
```
#At the end of each epoch, we need to reset the states before we go back to the␣
 ↪beginning
#of the text.
class ResetStatesCallback(keras.callbacks.Callback):
    def on_epoch_begin(self, epoch, logs):
        self.model.reset_states()
```

[51]:
```
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam")
steps_per_epoch = train_size // batch_size // n_steps
history = model.fit(dataset, steps_per_epoch=steps_per_epoch, epochs=50,
                    callbacks=[ResetStatesCallback()])
```

```
Epoch 1/50
313/313 [==============================] - 37s 117ms/step - loss: 2.6141
Epoch 2/50
313/313 [==============================] - 34s 109ms/step - loss: 2.2166
Epoch 3/50
313/313 [==============================] - 34s 108ms/step - loss: 2.4933
Epoch 4/50
313/313 [==============================] - 33s 106ms/step - loss: 2.4647
Epoch 5/50
313/313 [==============================] - 40s 129ms/step - loss: 2.1566
Epoch 6/50
313/313 [==============================] - 42s 135ms/step - loss: 2.1538
Epoch 7/50
313/313 [==============================] - 41s 131ms/step - loss: 2.0766
Epoch 8/50
313/313 [==============================] - 47s 150ms/step - loss: 1.9987
Epoch 9/50
313/313 [==============================] - 40s 127ms/step - loss: 1.9450
Epoch 10/50
313/313 [==============================] - 43s 137ms/step - loss: 1.9253
Epoch 11/50
313/313 [==============================] - 45s 142ms/step - loss: 1.8333
Epoch 12/50
313/313 [==============================] - 51s 163ms/step - loss: 1.7943
Epoch 13/50
313/313 [==============================] - 81s 260ms/step - loss: 1.7648
Epoch 14/50
313/313 [==============================] - 77s 246ms/step - loss: 1.7442
```

```
Epoch 15/50
313/313 [==============================] - 77s 247ms/step - loss: 1.7258
Epoch 16/50
313/313 [==============================] - 78s 250ms/step - loss: 1.7109
Epoch 17/50
313/313 [==============================] - 78s 250ms/step - loss: 1.7007
Epoch 18/50
313/313 [==============================] - 81s 259ms/step - loss: 1.6864
Epoch 19/50
313/313 [==============================] - 86s 273ms/step - loss: 1.6775
Epoch 20/50
313/313 [==============================] - 82s 261ms/step - loss: 1.6689
Epoch 21/50
313/313 [==============================] - 78s 249ms/step - loss: 1.6606
Epoch 22/50
313/313 [==============================] - 82s 263ms/step - loss: 1.6529
Epoch 23/50
313/313 [==============================] - 82s 261ms/step - loss: 1.6480
Epoch 24/50
313/313 [==============================] - 76s 244ms/step - loss: 1.6419
Epoch 25/50
313/313 [==============================] - 76s 242ms/step - loss: 1.6351
Epoch 26/50
313/313 [==============================] - 75s 241ms/step - loss: 1.6301
Epoch 27/50
313/313 [==============================] - 75s 241ms/step - loss: 1.6250
Epoch 28/50
313/313 [==============================] - 76s 243ms/step - loss: 1.6212
Epoch 29/50
313/313 [==============================] - 75s 240ms/step - loss: 1.6162
Epoch 30/50
313/313 [==============================] - 76s 243ms/step - loss: 1.6116
Epoch 31/50
313/313 [==============================] - 78s 248ms/step - loss: 1.6076
Epoch 32/50
313/313 [==============================] - 63s 201ms/step - loss: 1.6042
Epoch 33/50
313/313 [==============================] - 37s 119ms/step - loss: 1.6007
Epoch 34/50
313/313 [==============================] - 35s 110ms/step - loss: 1.5974
Epoch 35/50
313/313 [==============================] - 33s 104ms/step - loss: 1.5947
Epoch 36/50
313/313 [==============================] - 32s 103ms/step - loss: 1.5912
Epoch 37/50
313/313 [==============================] - 33s 106ms/step - loss: 1.5884
Epoch 38/50
313/313 [==============================] - 33s 107ms/step - loss: 1.5867
```

```
Epoch 39/50
313/313 [==============================] - 32s 104ms/step - loss: 1.5839
Epoch 40/50
313/313 [==============================] - 33s 104ms/step - loss: 1.5819
Epoch 41/50
313/313 [==============================] - 33s 105ms/step - loss: 1.5785
Epoch 42/50
313/313 [==============================] - 33s 105ms/step - loss: 1.5773
Epoch 43/50
313/313 [==============================] - 32s 102ms/step - loss: 1.5740
Epoch 44/50
313/313 [==============================] - 32s 101ms/step - loss: 1.5718
Epoch 45/50
313/313 [==============================] - 32s 102ms/step - loss: 1.5704
Epoch 46/50
313/313 [==============================] - 34s 107ms/step - loss: 1.5690
Epoch 47/50
313/313 [==============================] - 39s 126ms/step - loss: 1.5676
Epoch 48/50
313/313 [==============================] - 35s 111ms/step - loss: 1.5654
Epoch 49/50
313/313 [==============================] - 34s 107ms/step - loss: 1.5638
Epoch 50/50
313/313 [==============================] - 32s 103ms/step - loss: 1.5621
```

To use the model with different batch sizes, we need to create a stateless copy. We can get rid of dropout since it is only used during training:

[52]:
```
# Create a copy of this model that we can use to stateless data.
stateless_model = keras.models.Sequential([
    keras.layers.GRU(128, return_sequences=True, input_shape=[None, max_id]),
    keras.layers.GRU(128, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(max_id,
                                                    activation="softmax"))
])
```

To set the weights, we first need to build the model (so the weights get created):

[53]:
```
# build a model without estimation
stateless_model.build(tf.TensorShape([None, None, max_id]))
```

[54]:
```
# copy the weights from the stateful model to stateless model
stateless_model.set_weights(model.get_weights())
model = stateless_model
```

[55]:
```
tf.random.set_seed(42)
# run model
print(complete_text("t"))
```

```
ting;
do desire.

escalus:
no mouth, and fly very w
```

3 Sentiment Analysis

IMDb reviews dataset is the "hello world" of natural language processing, like MNIST for image recognition.

- 50,000 movie reviews in English (25,000 for training, 25,000 for testing)
- binary target for whether review is negative (0) or positive (1).

[4]:
```
tf.random.set_seed(42)
```

You can load the IMDB dataset easily:

[5]:
```
(X_train, y_test), (X_valid, y_test) = keras.datasets.imdb.load_data()
```

[6]:
```
X_train[0][:10]
```

[6]:
```
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65]
```

X_train is the preprocessed list integers, where each integer represents a word.

All punctuation was removed, and then words were converted to lowercase, split by spaces, and finally indexed by frequency (so low integers correspond to frequent words).

The integers 0, 1, and 2 are special: they represent the padding token, the start-of-sequence (SSS) token, and unknown words, respectively. If you want to visualize a review, you can decode it like this:

[7]:
```
word_index = keras.datasets.imdb.get_word_index()
# get word dictionary
id_to_word = {id_ + 3: word for word, id_ in word_index.items()}

for id_, token in enumerate(("<pad>", "<sos>", "<unk>")):
    #add these special symbols to the word dictionary
    id_to_word[id_] = token
" ".join([id_to_word[id_] for id_ in X_train[0][:10]])
```

[7]:
```
'<sos> this film was just brilliant casting location scenery story'
```

[ ]:
```

```

[8]:
```
#!pip install tensorflow-datasets
import tensorflow_datasets as tfds
datasets, info = tfds.load("imdb_reviews", as_supervised=True, with_info=True)
```

In a real project, you will have to preprocess the text yourself. You can do that using the same Tokenizer class we used earlier, but this time setting char_level=False. It will filter out a lot of characters, including most punctuation, line breaks, and tabs (you can change that).

The tokenizer will identify space as word boundary. This will work for English, but not for some other languages: Chinese (no spaces in sentence), Vietnamese (spaces between words). Even in English some words like "San Francisco" or "#ILoveDeepLearning." are hard to tokenize properly.

Google's SentencePiece project provides unsupervised learning technique to tokenize and detokenize text at the subword level in a language-independent way, treating spaces like other characters.

With this approach, even if your model encounters a word it has never seen before, it can still guess what it means. For example, it may never have seen the word "smartest" during training, but perhaps it learned the word "smart" and it also learned that the suffix "est" means "the most," so it can infer the meaning of "smartest."

Another option was proposed in an earlier paper by Rico Sennrich et al. that explored other ways of creating subword encodings (e.g., using byte pair encoding). Last but not least, the TensorFlow team released the TF.Text library in June 2019, which implements various tokenization strategies, including WordPiece7 (a variant of byte pair encoding).

If you want to deploy your model to a mobile device or a web browser, and you don't want to have to write a different preprocessing function every time, then you will want to handle preprocessing using only TensorFlow operations, so it can be included in the model itself. Let's see how. First, let's load the original IMDb reviews, as text (byte strings), using TensorFlow Datasets (introduced in Chapter 13):

[9]:
```
# subsets
datasets.keys()
```

[9]:
```
dict_keys(['test', 'train', 'unsupervised'])
```

[10]:
```
# get training and testing
train_size = info.splits["train"].num_examples
test_size = info.splits["test"].num_examples
```

[11]:
```
train_size, test_size
```

[11]:
```
(25000, 25000)
```

[15]:
```
for item in datasets["train"].take(3):
    print(item)
```

```
(<tf.Tensor: shape=(), dtype=string, numpy=b"This was an absolutely terrible
movie. Don't be lured in by Christopher Walken or Michael Ironside. Both are
great actors, but this must simply be their worst role in history. Even their
great acting could not redeem this movie's ridiculous storyline. This movie is
an early nineties US propaganda piece. The most pathetic scenes were those when
the Columbian rebels were making their cases for revolutions. Maria Conchita
```

```
Alonso appeared phony, and her pseudo-love affair with Walken was nothing but a
pathetic emotional plug in a movie that was devoid of any real meaning. I am
disappointed that there are movies like this, ruining actor's like Christopher
Walken's good name. I could barely sit through it.">, <tf.Tensor: shape=(),
dtype=int64, numpy=0>)
(<tf.Tensor: shape=(), dtype=string, numpy=b'I have been known to fall asleep
during films, but this is usually due to a combination of things including,
really tired, being warm and comfortable on the sette and having just eaten a
lot. However on this occasion I fell asleep because the film was rubbish. The
plot development was constant. Constantly slow and boring. Things seemed to
happen, but with no explanation of what was causing them or why. I admit, I may
have missed part of the film, but i watched the majority of it and everything
just seemed to happen of its own accord without any real concern for anything
else. I cant recommend this film at all.'>, <tf.Tensor: shape=(), dtype=int64,
numpy=0>)
(<tf.Tensor: shape=(), dtype=string, numpy=b'Mann photographs the Alberta Rocky
Mountains in a superb fashion, and Jimmy Stewart and Walter Brennan give
enjoyable performances as they always seem to do. <br /><br />But come on
Hollywood - a Mountie telling the people of Dawson City, Yukon to elect
themselves a marshal (yes a marshal!) and to enforce the law themselves, then
gunfighters battling it out on the streets for control of the town? <br /><br
/>Nothing even remotely resembling that happened on the Canadian side of the
border during the Klondike gold rush. Mr. Mann and company appear to have
mistaken Dawson City for Deadwood, the Canadian North for the American Wild
West.<br /><br />Canadian viewers be prepared for a Reefer Madness type of
enjoyable howl with this ludicrous plot, or, to shake your head in disgust.'>,
<tf.Tensor: shape=(), dtype=int64, numpy=0>)
```

[64]:
```python
# get two batches and one elements from each batch
for X_batch, y_batch in datasets["train"].batch(2).take(1):
    for review, label in zip(X_batch.numpy(), y_batch.numpy()):
        print("Review:", review.decode("utf-8")[:200], "...")
        print("Label:", label, "= Positive" if label else "= Negative")
        print()
```

```
Review: This was an absolutely terrible movie. Don't be lured in by Christopher
Walken or Michael Ironside. Both are great actors, but this must simply be their
worst role in history. Even their great acting  …
Label: 0 = Negative

Review: I have been known to fall asleep during films, but this is usually due
to a combination of things including, really tired, being warm and comfortable
on the sette and having just eaten a lot. However  …
Label: 0 = Negative
```

[65]:
```
# We need to clean this
def preprocess(X_batch, y_batch):
    # get first 300 words
    X_batch = tf.strings.substr(X_batch, 0, 300)
    X_batch = tf.strings.regex_replace(X_batch, rb"<br\s*/?>", b" ")
    X_batch = tf.strings.regex_replace(X_batch, b"[^a-zA-Z']", b" ")
    X_batch = tf.strings.split(X_batch)
    return X_batch.to_tensor(default_value=b"<pad>"), y_batch
```

- keeping only the first 300 characters of each review to speed up training. We can figure the sentiment fast.
- Use regular expressions to replace tags with spaces.
- Replace any characters other than letters and quotes with spaces.

Finally, the preprocess() function splits the reviews by the spaces, which returns a ragged tensor, and then dense tensor, and then padding all reviews with the padding token “” so that they all have the same length.

[66]:
```
preprocess(X_batch, y_batch)
```

[66]:
```
(<tf.Tensor: shape=(2, 53), dtype=string, numpy=
 array([[b'This', b'was', b'an', b'absolutely', b'terrible', b'movie',
         b"Don't", b'be', b'lured', b'in', b'by', b'Christopher',
         b'Walken', b'or', b'Michael', b'Ironside', b'Both', b'are',
         b'great', b'actors', b'but', b'this', b'must', b'simply', b'be',
         b'their', b'worst', b'role', b'in', b'history', b'Even',
         b'their', b'great', b'acting', b'could', b'not', b'redeem',
         b'this', b"movie's", b'ridiculous', b'storyline', b'This',
         b'movie', b'is', b'an', b'early', b'nineties', b'US',
         b'propaganda', b'pi', b'<pad>', b'<pad>', b'<pad>'],
        [b'I', b'have', b'been', b'known', b'to', b'fall', b'asleep',
         b'during', b'films', b'but', b'this', b'is', b'usually', b'due',
         b'to', b'a', b'combination', b'of', b'things', b'including',
         b'really', b'tired', b'being', b'warm', b'and', b'comfortable',
         b'on', b'the', b'sette', b'and', b'having', b'just', b'eaten',
         b'a', b'lot', b'However', b'on', b'this', b'occasion', b'I',
         b'fell', b'asleep', b'because', b'the', b'film', b'was',
         b'rubbish', b'The', b'plot', b'development', b'was', b'constant',
         b'Cons']], dtype=object)>,
 <tf.Tensor: shape=(2,), dtype=int64, numpy=array([0, 0])>)
```

We will construct the vocabulary by going through the whole training set once, applying our preprocess() function, and using a Counter to count the number of occurrences of each word:

[67]:
```
from collections import Counter
# get frequency counter
vocabulary = Counter()
for X_batch, y_batch in datasets["train"].batch(32).map(preprocess):
```

```
    for review in X_batch:
        vocabulary.update(list(review.numpy()))
```

[68]:
```
print(f"Most common {vocabulary.most_common()[:3]}, the least common␣
↪{vocabulary.most_common()[-3:]}")
#least common
```

```
Most common [(b'<pad>', 214309), (b'the', 61137), (b'a', 38564)], the least
common [(b'xico', 1), (b"'dogma'", 1), (b"end'", 1)]
```

[69]:
```
len(vocabulary)
```

[69]:
```
53893
```

We don't need all words, let's keep 10,000 most common:

[70]:
```
vocab_size = 10000
truncated_vocabulary = [
    word for word, count in vocabulary.most_common()[:vocab_size]]
```

[71]:
```
word_to_id = {word: index for index, word in enumerate(truncated_vocabulary)}
for word in b"This movie was faaaaaantastic".split():
    print(word_to_id.get(word) or vocab_size)
```

```
22
12
11
10000
```

Now we need to add a preprocessing step to replace each word with its ID (i.e., its index in the vocabulary). We will create a lookup table for this, using 1,000 out-of-vocabulary (oov) buckets. The oov are used to account for the differences in vocabularly between training and testing date. We expect to have maximum 1000 words in testing data that are not found in training data. We will use oov for these words, if the actual number of new words will be higher, the model will have an index conflic which will decrease it's efficiency.

[72]:
```
# define vocabylary: list of all possible categories
words = tf.constant(truncated_vocabulary)
# tensor corresponding to indexes of word IDs
word_ids = tf.range(len(truncated_vocabulary), dtype=tf.int64)
#Create an initializer for the lookup table, passing it the list of categories␣
↪and their corresponding indices.

# This is basically Tensorflow dictionary
vocab_init = tf.lookup.KeyValueTensorInitializer(words, word_ids)
num_oov_buckets = 1000
table = tf.lookup.StaticVocabularyTable(vocab_init, num_oov_buckets)
```

We can then use this table to look up the IDs of a few words:

[73]:
```
table.lookup(tf.constant([b"This movie was faaaaaantastic".split()]))
```

[73]:
```
<tf.Tensor: shape=(1, 4), dtype=int64, numpy=array([[   22,    12,    11,
10053]])>
```

The words "this," "movie," and "was" were found in the table, so their IDs are lower than 10,000, while the word "faaaaaantastic" was not found, so it was mapped to one of the oov buckets, with an ID greater than 10,000.

[74]:
```
# Fundction that return codes for the words in X_batch
def encode_words(X_batch, y_batch):
    return table.lookup(X_batch), y_batch
#batch the reviews and then convert them to short sequences of words using the␣
 ↪preprocess() function
train_set = datasets["train"].repeat().batch(32).map(preprocess)
# Encode words with numeric codes
train_set = train_set.map(encode_words).prefetch(1)
```

[75]:
```
# Look at the data shape
for X_batch, y_batch in train_set.take(1):
    print(X_batch)
    print(y_batch)
```

```
tf.Tensor(
[[  22   11   28 …    0    0    0]
 [   6   21   70 …    0    0    0]
 [4099 6881    1 …    0    0    0]
 …
 [  22   12  118 …  331 1047    0]
 [1757 4101  451 …    0    0    0]
 [3365 4392    6 …    0    0    0]], shape=(32, 60), dtype=int64)
tf.Tensor([0 0 0 1 1 1 0 0 0 0 0 1 1 0 1 0 1 1 1 0 1 1 1 1 1 0 0 0 1 0 0 0],
shape=(32,), dtype=int64)
```

[ ]:
```

```

4 Embeddings

The first layer is an Embedding layer, which will convert word IDs into embeddings – relationships between words (see chapter 13 for more details). The embedding matrix needs to have one row per word ID (vocab_size + num_oov_buckets) and one column per embedding dimension (this example uses 128 dimensions, but this is a hyperparameter you could tune).

The inputs of the model will be 2D tensors of shape [batch size, time steps], the output of the Embedding layer will be a 3D tensor of shape [batch size, time steps, embedding size].

The Embedding layer can be understood as a lookup table that maps from integer indices (which stand for specific words) to dense vectors (their embeddings). The dimensionality (or width) of the

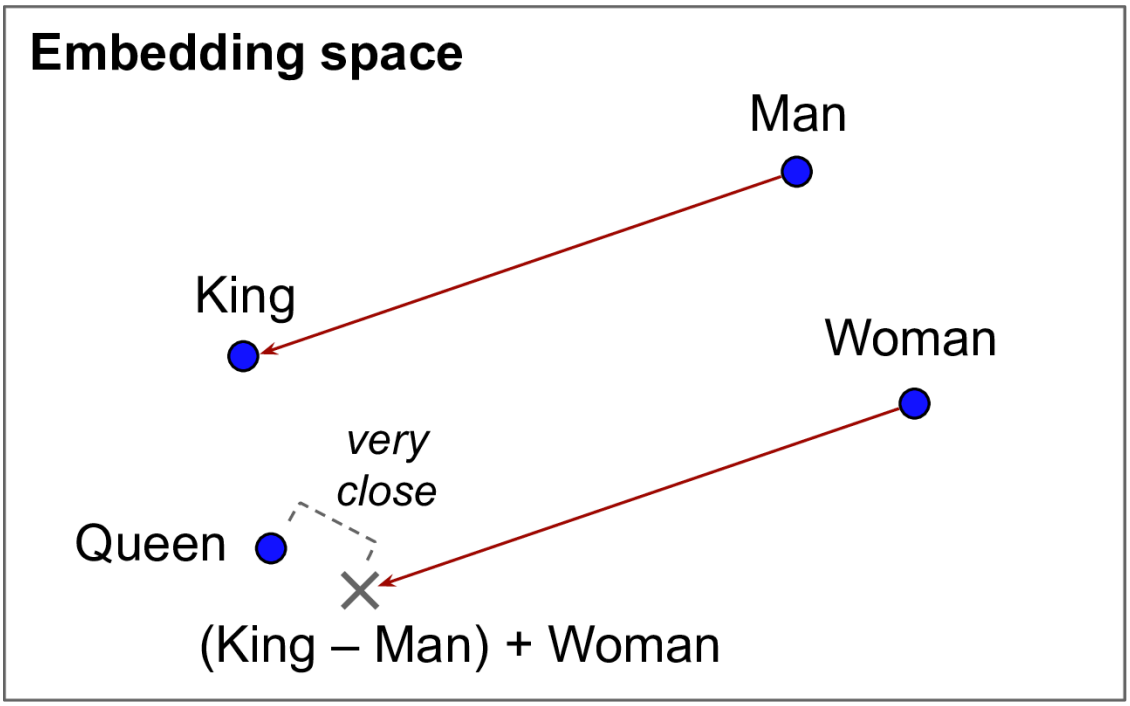
embedding is a parameter works in the same way as the number of neurons in a Dense layer.

The idea of using vectors to represent words dates back to the 1960s. The idea is to reduce the dimensionality of language, while preserving it's ability to describe key concepts.

First,mneural network predicts the words near any given word, and obtains connections between them. For example, synonyms had very close embeddings, and semantically related words such as France, Spain, and Italy ended up clustered together.

It's not just about proximity, though: word embeddings were also organized along meaningful axes in the embedding space. Here is a famous example: if you compute $King–Man + Woman \simeq Female\ King \simeq Queen$ $Madrid–Spain + France \simeq French\ Capital \simeq Paris$

The word embeddings encode the concept of gender and capital cities



Next we add two GRU layers with the second one returning only the output of the last time step.

The output layer a single neuron using the sigmoid activation function to output the estimated probability that the review expresses a positive sentiment regarding the movie.

[77]:
```
# Run model
embed_size = 128
model = keras.models.Sequential([
    keras.layers.Embedding(vocab_size + num_oov_buckets, embed_size,
                           mask_zero=True, # not shown in the book
                           input_shape=[None]),
    keras.layers.GRU(128, return_sequences=True),
    keras.layers.GRU(128),
    keras.layers.Dense(1, activation="sigmoid")
```

```
])
model.compile(loss="binary_crossentropy", optimizer="adam",␣
 ↪metrics=["accuracy"])
history = model.fit(train_set, steps_per_epoch=train_size // 32, epochs=5)
```

```
Epoch 1/5
781/781 [==============================] - 59s 75ms/step - loss: 0.5305 -
accuracy: 0.7282
Epoch 2/5
781/781 [==============================] - 59s 75ms/step - loss: 0.3459 -
accuracy: 0.8554
Epoch 3/5
781/781 [==============================] - 62s 80ms/step - loss: 0.1913 -
accuracy: 0.9319
Epoch 4/5
781/781 [==============================] - 58s 75ms/step - loss: 0.1341 -
accuracy: 0.9535
Epoch 5/5
781/781 [==============================] - 59s 76ms/step - loss: 0.1011 -
accuracy: 0.9624
```

The model will need to ignore padding tokens (reviews less than 300 characters). This trivial task wastes time and accuracy. If we add *mask_zero=True* they will be ignored by all downstream layers.

If you do it manually: the Embedding layer creates a mask tensor equal to K.not_equal(inputs, 0) (where K = keras.backend): it is a Boolean tensor with the same shape as the inputs, and it is equal to False anywhere the word IDs are 0, or True otherwise.

This mask tensor is then automatically propagated by the model to all subsequent layers.

Both GRU layers will receive this mask automatically, but since the second GRU layer does not return sequences (it only returns the output of the last time step), the mask will not be transmitted to the Dense layer.

Each layer may handle the mask differently, but in general they simply ignore masked time steps (i.e., time steps for which the mask is False). For example, when a recurrent layer encounters a masked time step, it simply copies the output from the previous time step. If the mask propagates all the way to the output (in models that output sequences, which is not the case in this example), then it will be applied to the losses as well, so the masked time steps will not contribute to the loss (their loss will be 0).

All layers that receive the mask must support masking (or else an exception will be raised). This includes all recurrent layers, as well as the TimeDistributed layer and a few other layers.

Any layer that supports masking must have a supports_masking attribute equal to True.

Using masking layers and automatic mask propagation works best for simple Sequential models. It will not always work for more complex models, such as when you need to mix Conv1D layers with recurrent layers.

After training for a few epochs, this model will become quite good at judging whether a review is positive or not.

t'simpressive that the model is able to learn useful word embeddings based on just 25,000 movie reviews.

Imagine how good the embeddings would be if we had billions of reviews to train on! Unfortunately we don't, but perhaps we can reuse word embeddings trained on some other large text corpus (e.g., Wikipedia articles), even if it is not composed of movie reviews? After all, the word "amazing" generally has the same meaning whether you use it to talk about movies or anything else. Moreover, perhaps embeddings would be useful for sentiment analysis even if they were trained on another task: since words like "awesome" and "amazing" have a similar meaning, they will likely cluster in the embedding space even for other tasks (e.g., predicting the next word in a sentence). If all positive words and all negative words form clusters, then this will be helpful for sentiment analysis. So instead of using so many parameters to learn word embeddings, let's see if we can't just reuse pretrained embeddings.

4.1 Reusing Pretrained Embeddings

We reuse a sentence encoder: it takes strings as input and encodes each one as a single vector (in this case, a 50-dimensional vector).

It parses the string (splitting words on spaces) and embeds each word using an embedding matrix that was pretrained on a huge corpus: the Google News 7B corpus (seven billion words long!).

Then it computes the mean of all the word embeddings, and the result is the sentence embedding.

We can then add two simple Dense layers to create a good sentiment analysis model. By default, a hub.KerasLayer is not trainable, but you can set trainable=True when creating it to change that so that you can fine-tune it for your task.

[83]:
```
tf.random.set_seed(42)
```

[84]:
```
TFHUB_CACHE_DIR = os.path.join(os.curdir, "my_tfhub_cache")
os.environ["TFHUB_CACHE_DIR"] = TFHUB_CACHE_DIR
```

[85]:
```
!pip install tensorflow_hub
import tensorflow_hub as hub

model = keras.Sequential([
    hub.KerasLayer("https://tfhub.dev/google/tf2-preview/nnlm-en-dim50/1",
                   dtype=tf.string, input_shape=[], output_shape=[50]),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dense(1, activation="sigmoid")
])
model.compile(loss="binary_crossentropy", optimizer="adam",
              metrics=["accuracy"])
```

```
Requirement already satisfied: tensorflow_hub in
/Users/ir177/opt/anaconda3/lib/python3.8/site-packages (0.10.0)
Requirement already satisfied: protobuf>=3.8.0 in
```

```
/Users/ir177/opt/anaconda3/lib/python3.8/site-packages (from tensorflow_hub)
(3.12.4)
Requirement already satisfied: numpy>=1.12.0 in
/Users/ir177/opt/anaconda3/lib/python3.8/site-packages (from tensorflow_hub)
(1.18.5)
Requirement already satisfied: six>=1.9 in
/Users/ir177/opt/anaconda3/lib/python3.8/site-packages (from
protobuf>=3.8.0->tensorflow_hub) (1.15.0)
Requirement already satisfied: setuptools in
/Users/ir177/opt/anaconda3/lib/python3.8/site-packages (from
protobuf>=3.8.0->tensorflow_hub) (49.2.0.post20200714)
```

[86]:
```
for dirpath, dirnames, filenames in os.walk(TFHUB_CACHE_DIR):
    for filename in filenames:
        print(os.path.join(dirpath, filename))
```

```
./my_tfhub_cache/82c4aaf4250ffb09088bd48368ee7fd00e5464fe.descriptor.txt
./my_tfhub_cache/82c4aaf4250ffb09088bd48368ee7fd00e5464fe/saved_model.pb
./my_tfhub_cache/82c4aaf4250ffb09088bd48368ee7fd00e5464fe/variables/variables.da
ta-00000-of-00001
./my_tfhub_cache/82c4aaf4250ffb09088bd48368ee7fd00e5464fe/variables/variables.in
dex
./my_tfhub_cache/82c4aaf4250ffb09088bd48368ee7fd00e5464fe/assets/tokens.txt
```

[87]:
```
import tensorflow_datasets as tfds

datasets, info = tfds.load("imdb_reviews", as_supervised=True, with_info=True)
train_size = info.splits["train"].num_examples
batch_size = 32
train_set = datasets["train"].repeat().batch(batch_size).prefetch(1)
history = model.fit(train_set, steps_per_epoch=train_size // batch_size,␣
 ↪epochs=5)
```

```
Epoch 1/5
781/781 [==============================] - 34s 43ms/step - loss: 0.5460 -
accuracy: 0.7267
Epoch 2/5
781/781 [==============================] - 35s 44ms/step - loss: 0.5129 -
accuracy: 0.7495
Epoch 3/5
781/781 [==============================] - 35s 45ms/step - loss: 0.5082 -
accuracy: 0.7530
Epoch 4/5
781/781 [==============================] - 35s 45ms/step - loss: 0.5047 -
accuracy: 0.7533
Epoch 5/5
781/781 [==============================] - 35s 45ms/step - loss: 0.5015 -
accuracy: 0.7560
```

4.2 Automatic Translation

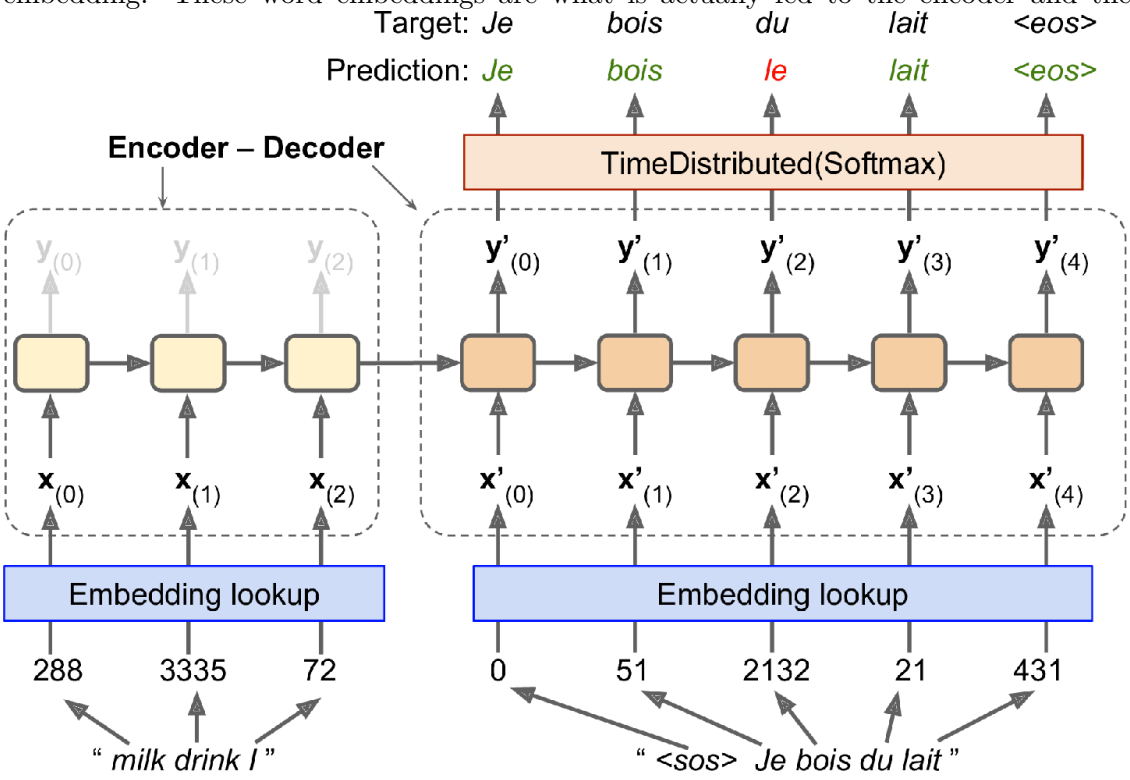
Let's try to translated English sentence to French.

The English sentences are fed to the encoder, and the decoder outputs the French translations.

The French translations are also used as inputs to the decoder, but shifted back by one step. Because it not only trasnlate the sentence, but also tried to predict the fitting French word next in the sentence. The first word is the start-of-sequence (SOS) token and the end is end-of-sequence (EOS) token.

The English sentence is reversed before they are fed to the encoder: "I drink milk" -> "milk drink I." In translation the key informaiton often occurs in the end of the sentence, so decoded needs to translate it first.

Each word is its ID (e.g., 288 for the word "milk"). Next, an embedding layer returns the word embedding. These word embeddings are what is actually fed to the encoder and the decoder.



At each step, the decoder outputs a score for each word in the output vocabulary (i.e.,French), and then the softmax layer turns these scores into probabilities.

For example, at the first step the word "Je" may have a probability of 20%, "Tu" may have a probability of 1%, and so on. The word with the highest probability is output.

Few details:

- Sames sentences use different number of words in different languages. We group sentences into buckets of similar lengths (e.g., a bucket for the 1- to 6-word sentences, another for the 7- to 12-word sentences, and so on), using padding for the shorter sequences to ensure all sentences in a bucket have the same length: For example, "I drink milk" becomes " milk drink I."

• We want to ignore any output past the EOS token, so these tokens should not contribute to the loss (they must be masked out). For example, if the model outputs "Je bois du lait oui," the loss for the last word should be ignored. • When the output vocabulary is large the outputting a probability for each and every possible word would be terribly slow. 50,000 words -> 50,000 dimensional output vector of probabilities. * One way to speed it up is look at the probability of a correct word and a random sample of incorrect words for accuracy calculation.

[88]:
```
tf.random.set_seed(42)
```

[89]:
```
vocab_size = 100
embed_size = 10
```

[90]:
```
import tensorflow_addons as tfa
# flexible model for any lengths
encoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
decoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
#
sequence_lengths = keras.layers.Input(shape=[], dtype=np.int32)
# First layer is word embeddings
embeddings = keras.layers.Embedding(vocab_size, embed_size)
# inputs and outpus
encoder_embeddings = embeddings(encoder_inputs)
decoder_embeddings = embeddings(decoder_inputs)
# hidden LSTM layers
encoder = keras.layers.LSTM(512, return_state=True)
# Convert hidden layers outputs to embeddings
encoder_outputs, state_h, state_c = encoder(encoder_embeddings)
encoder_state = [state_h, state_c]

sampler = tfa.seq2seq.sampler.TrainingSampler()
# Next decode from embeddings to French words translations
decoder_cell = keras.layers.LSTMCell(512)
# French words layers, here we both translate words and predict the next word␣
 ↪based on the other French word in
# a sentence
output_layer = keras.layers.Dense(vocab_size)
decoder = tfa.seq2seq.basic_decoder.BasicDecoder(decoder_cell, sampler,
                                                 output_layer=output_layer)
# save output from the decode rnn
final_outputs, final_state, final_sequence_lengths = decoder(
    decoder_embeddings, initial_state=encoder_state,
    sequence_length=sequence_lengths)
# pick the word with the highest probability
Y_proba = tf.nn.softmax(final_outputs.rnn_output)
# construct a model
model = keras.models.Model(
    inputs=[encoder_inputs, decoder_inputs, sequence_lengths],
```

```
    outputs=[Y_proba])
```

Set return_state=True when creating the LSTM layer to get the final hidden state and pass it to the decoder (LTSM returns two hidden states short term and long term).

The TrainingSampler is one of several samplers available in TensorFlow Addons: their role is to tell the decoder at each step what it should pretend the previous output was. To predict the next word based on the target language model.

[91]:
```
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam")
```

[92]:
```
# Test model on random word IDs using 1000 15-word sentences
X = np.random.randint(100, size=10*1000).reshape(1000, 10)
Y = np.random.randint(100, size=15*1000).reshape(1000, 15)
X_decoder = np.c_[np.zeros((1000, 1)), Y[:, :-1]]
seq_lengths = np.full([1000], 15)

history = model.fit([X, X_decoder, seq_lengths], Y, epochs=2)
```

```
Epoch 1/2
32/32 [==============================] - 4s 130ms/step - loss: 4.6052
Epoch 2/2
32/32 [==============================] - 3s 90ms/step - loss: 4.6027
```

4.2.1 Bidirectional Recurrent Layers

RNN looks in the past and generates output for the future. It works well for time-series forecasting, but not for translation, where you need to go back and edit knowing what word would go next.

For example, consider: “the Queen of the United Kingdom,” “the queen of hearts,” and “the queen bee”: to properly encode the word “queen,” you need to look ahead.

To implement this, run two recurrent layers on the same inputs, one reading the words from left to right and the other reading them from right to left. Then concatenate the outputs are each steps.

To implement a bidirectional recurrent layer in Keras, wrap a recurrent layer in a keras.layers.Bidirectional layer.

[93]:
```
model = keras.models.Sequential([
    keras.layers.GRU(10, return_sequences=True, input_shape=[None, 10]),
    keras.layers.Bidirectional(keras.layers.GRU(10, return_sequences=True))
])

model.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
gru_16 (GRU)                 (None, None, 10)          660
```

```
_________________________________________________________________
bidirectional (Bidirectional (None, None, 20)          1320
=================================================================
Total params: 1,980
Trainable params: 1,980
Non-trainable params: 0
_________________________________________________________________
```

5 Exercise solutions

5.1 1. to 7.

See Appendix A.

5.2 8.

Exercise: Embedded Reber grammars *were used by Hochreiter and Schmidhuber in their paper about LSTMs. They are artificial grammars that produce strings such as "BPBTSXXVPSEPE." Check out Jenny Orr's nice introduction to this topic. Choose a particular embedded Reber grammar (such as the one represented on Jenny Orr's page), then train an RNN to identify whether a string respects that grammar or not. You will first need to write a function capable of generating a training batch containing about 50% strings that respect the grammar, and 50% that don't.*

First we need to build a function that generates strings based on a grammar. The grammar will be represented as a list of possible transitions for each state. A transition specifies the string to output (or a grammar to generate it) and the next state.

[ ]:
```
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)
```

Let's generate a few strings based on the default Reber grammar:

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

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

Looks good. Now let's generate a few strings based on the embedded Reber grammar:

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

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

Okay, now we need a function to generate strings that do not respect the grammar. We could generate a random string, but the task would be a bit too easy, so instead we will generate a string that respects the grammar, and we will corrupt it by changing just one character:

[ ]:
```
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:]
```

Let's look at a few corrupted strings:

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

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

We cannot feed strings directly to an RNN, so we need to encode them somehow. One option would be to one-hot encode each character. Another option is to use embeddings. Let's go for the second option (but since there are just a handful of characters, one-hot encoding would probably be a good option as well). For embeddings to work, we need to convert each string into a sequence of character IDs. Let's write a function for that, using each character's index in the string of possible characters "BEPSTVX":

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

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

We can now generate the dataset, with 50% good strings, and 50% bad strings:

[ ]:
```
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
```

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

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

Let's take a look at the first training sequence:

[ ]:
```
X_train[0]
```

What classes does it belong to?

[ ]:
```
y_train[0]
```

Perfect! We are ready to create the RNN to identify good strings. We build a simple sequence binary classifier:

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

embedding_size = 5

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

Now let's test our RNN on two tricky strings: the first one is bad while the second one is good. They only differ by the second to last character. If the RNN gets this right, it shows that it managed to notice the pattern that the second letter should always be equal to the second to last letter. That requires a fairly long short-term memory (which is the reason why we used a GRU cell).

[ ]:
```
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]))
```

Ta-da! It worked fine. The RNN found the correct answers with very high confidence. :)

5.3 9.

Exercise: Train an Encoder–Decoder model that can convert a date string from one format to another (e.g., from "April 22, 2019" to "2019-04-22").

Let's start by creating the dataset. We will use random days between 1000-01-01 and 9999-12-31:

[ ]:
```
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
```

Here are a few random dates, displayed in both the input format and the target format:

[ ]:
```
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]))
```

Let's get the list of all possible characters in the inputs:

[ ]:
```
INPUT_CHARS = "".join(sorted(set("".join(MONTHS)))) + "01234567890, "
INPUT_CHARS
```

And here's the list of possible characters in the outputs:

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

Let's write a function to convert a string to a list of character IDs, as we did in the previous exercise:

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

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

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

[ ]:
```
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)
```

[ ]:
```
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)
```

[ ]:
```
Y_train[0]
```

5.3.1 First version: a very basic seq2seq model

Let's first try the simplest possible model: we feed in the input sequence, which first goes through the encoder (an embedding layer followed by a single LSTM layer), which outputs a vector, then it

goes through a decoder (a single LSTM layer, followed by a dense output layer), which outputs a sequence of vectors, each representing the estimated probabilities for all possible output character.

Since the decoder expects a sequence as input, we repeat the vector (which is output by the decoder) as many times as the longest possible output sequence.

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

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

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

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

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

optimizer = 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))
```

Looks great, we reach 100% validation accuracy! Let's use the model to make some predictions. We will need to be able to convert a sequence of character IDs to a readable string:

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

Now we can use the model to convert some dates

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

```
[ ]: ids = model.predict_classes(X_new)
     for date_str in ids_to_date_strs(ids):
         print(date_str)
```

Perfect! :)

However, since the model was only trained on input strings of length 18 (which is the length of the longest date), it does not perform well if we try to use it to make predictions on shorter sequences:

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

```
[ ]: ids = model.predict_classes(X_new)
     for date_str in ids_to_date_strs(ids):
         print(date_str)
```

Oops! We need to ensure that we always pass sequences of the same length as during training, using padding if necessary. Let's write a little helper function for that:

```
[ ]: 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_classes(X)
         return ids_to_date_strs(ids)
```

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

Cool! Granted, there are certainly much easier ways to write a date conversion tool (e.g., using regular expressions or even basic string manipulation), but you have to admit that using neural networks is way cooler. ;-)

However, real-life sequence-to-sequence problems will usually be harder, so for the sake of completeness, let's build a more powerful model.

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

Instead of feeding the decoder a simple repetition of the encoder's output vector, we can feed it the target sequence, shifted by one time step to the right. This way, at each time step the decoder will know what the previous target character was. This should help is tackle more complex sequence-to-sequence problems.

Since the first output character of each target sequence has no previous character, we will need a new token to represent the start-of-sequence (sos).

During inference, we won't know the target, so what will we feed the decoder? We can just predict one character at a time, starting with an sos token, then feeding the decoder all the characters that were predicted so far (we will look at this in more details later in this notebook).

But if the decoder's LSTM expects to get the previous target as input at each step, how shall we pass it it the vector output by the encoder? Well, one option is to ignore the output vector, and instead use the encoder's LSTM state as the initial state of the decoder's LSTM (which requires that encoder's LSTM must have the same number of units as the decoder's LSTM).

Now let's create the decoder's inputs (for training, validation and testing). The sos token will be represented using the last possible output character's ID + 1.

[ ]:
```
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)
```

Let's take a look at the decoder's training inputs:

[ ]:
```
X_train_decoder
```

Now let's build the model. It's not a simple sequential model anymore, so let's use the functional API:

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

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

encoder_input = keras.layers.Input(shape=[None], dtype=tf.int32)
encoder_embedding = keras.layers.Embedding(
    input_dim=len(INPUT_CHARS) + 1,
    output_dim=encoder_embedding_size)(encoder_input)
_, encoder_state_h, encoder_state_c = keras.layers.LSTM(
    lstm_units, return_state=True)(encoder_embedding)
encoder_state = [encoder_state_h, encoder_state_c]

decoder_input = keras.layers.Input(shape=[None], dtype=tf.int32)
decoder_embedding = keras.layers.Embedding(
    input_dim=len(OUTPUT_CHARS) + 2,
    output_dim=decoder_embedding_size)(decoder_input)
decoder_lstm_output = keras.layers.LSTM(lstm_units, return_sequences=True)(
```

```
    decoder_embedding, initial_state=encoder_state)
decoder_output = keras.layers.Dense(len(OUTPUT_CHARS) + 1,
                                    activation="softmax")(decoder_lstm_output)

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

optimizer = 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))
```

This model also reaches 100% validation accuracy, but it does so even faster.

Let's once again use the model to make some predictions. This time we need to predict characters one by one.

[ ]:
```
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:])
```

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

Works fine! :)

5.3.3 Third version: using TF-Addons's seq2seq implementation

Let's build exactly the same model, but using TF-Addon's seq2seq API. The implementation below is almost very similar to the TFA example higher in this notebook, except without the model input to specify the output sequence length, for simplicity (but you can easily add it back in if you need it for your projects, when the output sequences have very different lengths).

[ ]:
```
import tensorflow_addons as tfa

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

encoder_embedding_size = 32
```

```
decoder_embedding_size = 32
units = 128

encoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
decoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
sequence_lengths = keras.layers.Input(shape=[], dtype=np.int32)

encoder_embeddings = keras.layers.Embedding(
    len(INPUT_CHARS) + 1, encoder_embedding_size)(encoder_inputs)

decoder_embedding_layer = keras.layers.Embedding(
    len(INPUT_CHARS) + 2, decoder_embedding_size)
decoder_embeddings = decoder_embedding_layer(decoder_inputs)

encoder = keras.layers.LSTM(units, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_embeddings)
encoder_state = [state_h, state_c]

sampler = tfa.seq2seq.sampler.TrainingSampler()

decoder_cell = keras.layers.LSTMCell(units)
output_layer = keras.layers.Dense(len(OUTPUT_CHARS) + 1)

decoder = tfa.seq2seq.basic_decoder.BasicDecoder(decoder_cell,
                                                 sampler,
                                                 output_layer=output_layer)
final_outputs, final_state, final_sequence_lengths = decoder(
    decoder_embeddings,
    initial_state=encoder_state)
Y_proba = keras.layers.Activation("softmax")(final_outputs.rnn_output)

model = keras.models.Model(inputs=[encoder_inputs, decoder_inputs],
                           outputs=[Y_proba])
optimizer = keras.optimizers.Nadam()
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])
history = model.fit([X_train, X_train_decoder], Y_train, epochs=15,
                    validation_data=([X_valid, X_valid_decoder], Y_valid))
```

And once again, 100% validation accuracy! To use the model, we can just reuse the `predict_date_strs()` function:

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

However, there's a much more efficient way to perform inference. Until now, during inference, we've run the model once for each new character. Instead, we can create a new decoder, based on the previously trained layers, but using a `GreedyEmbeddingSampler` instead of a `TrainingSampler`.

At each time step, the `GreedyEmbeddingSampler` will compute the argmax of the decoder's outputs, and run the resulting token IDs through the decoder's embedding layer. Then it will feed the resulting embeddings to the decoder's LSTM cell at the next time step. This way, we only need to run the decoder once to get the full prediction.

[ ]:
```
inference_sampler = tfa.seq2seq.sampler.GreedyEmbeddingSampler(
    embedding_fn=decoder_embedding_layer)
inference_decoder = tfa.seq2seq.basic_decoder.BasicDecoder(
    decoder_cell, inference_sampler, output_layer=output_layer,
    maximum_iterations=max_output_length)
batch_size = tf.shape(encoder_inputs)[:1]
start_tokens = tf.fill(dims=batch_size, value=sos_id)
final_outputs, final_state, final_sequence_lengths = inference_decoder(
    start_tokens,
    initial_state=encoder_state,
    start_tokens=start_tokens,
    end_token=0)

inference_model = keras.models.Model(inputs=[encoder_inputs],
                                     outputs=[final_outputs.sample_id])
```

A few notes: * The `GreedyEmbeddingSampler` needs the `start_tokens` (a vector containing the start-of-sequence ID for each decoder sequence), and the `end_token` (the decoder will stop decoding a sequence once the model outputs this token). * We must set `maximum_iterations` when creating the `BasicDecoder`, or else it may run into an infinite loop (if the model never outputs the end token for at least one of the sequences). This would force you would to restart the Jupyter kernel. * The decoder inputs are not needed anymore, since all the decoder inputs are generated dynamically based on the outputs from the previous time step. * The model's outputs are `final_outputs.sample_id` instead of the softmax of `final_outputs.rnn_outputs`. This allows us to directly get the argmax of the model's outputs. If you prefer to have access to the logits, you can replace `final_outputs.sample_id` with `final_outputs.rnn_outputs`.

Now we can write a simple function that uses the model to perform the date format conversion:

[ ]:
```
def fast_predict_date_strs(date_strs):
    X = prepare_date_strs_padded(date_strs)
    Y_pred = inference_model.predict(X)
    return ids_to_date_strs(Y_pred)
```

[ ]:
```
fast_predict_date_strs(["July 14, 1789", "May 01, 2020"])
```

Let's check that it really is faster:

[ ]:
```
%timeit predict_date_strs(["July 14, 1789", "May 01, 2020"])
```

[ ]:
```
%timeit fast_predict_date_strs(["July 14, 1789", "May 01, 2020"])
```

That's more than a 10x speedup! And it would be even more if we were handling longer sequences.

5.3.4 Fourth version: using TF-Addons's seq2seq implementation with a scheduled sampler

Warning: due to a TF bug, this version only works using TensorFlow 2.2.

When we trained the previous model, at each time step t we gave the model the target token for time step t - 1. However, at inference time, the model did not get the previous target at each time step. Instead, it got the previous prediction. So there is a discrepancy between training and inference, which may lead to disappointing performance. To alleviate this, we can gradually replace the targets with the predictions, during training. For this, we just need to replace the `TrainingSampler` with a `ScheduledEmbeddingTrainingSampler`, and use a Keras callback to gradually increase the `sampling_probability` (i.e., the probability that the decoder will use the prediction from the previous time step rather than the target for the previous time step).

[ ]:
```
import tensorflow_addons as tfa

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

n_epochs = 20
encoder_embedding_size = 32
decoder_embedding_size = 32
units = 128

encoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
decoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
sequence_lengths = keras.layers.Input(shape=[], dtype=np.int32)

encoder_embeddings = keras.layers.Embedding(
    len(INPUT_CHARS) + 1, encoder_embedding_size)(encoder_inputs)

decoder_embedding_layer = keras.layers.Embedding(
    len(INPUT_CHARS) + 2, decoder_embedding_size)
decoder_embeddings = decoder_embedding_layer(decoder_inputs)

encoder = keras.layers.LSTM(units, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_embeddings)
encoder_state = [state_h, state_c]

sampler = tfa.seq2seq.sampler.ScheduledEmbeddingTrainingSampler(
    sampling_probability=0.,
    embedding_fn=decoder_embedding_layer)
# we must set the sampling_probability after creating the sampler
# (see https://github.com/tensorflow/addons/pull/1714)
sampler.sampling_probability = tf.Variable(0.)

decoder_cell = keras.layers.LSTMCell(units)
output_layer = keras.layers.Dense(len(OUTPUT_CHARS) + 1)
```

```
decoder = tfa.seq2seq.basic_decoder.BasicDecoder(decoder_cell,
                                                 sampler,
                                                 output_layer=output_layer)
final_outputs, final_state, final_sequence_lengths = decoder(
    decoder_embeddings,
    initial_state=encoder_state)
Y_proba = keras.layers.Activation("softmax")(final_outputs.rnn_output)

model = keras.models.Model(inputs=[encoder_inputs, decoder_inputs],
                           outputs=[Y_proba])
optimizer = keras.optimizers.Nadam()
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])

def update_sampling_probability(epoch, logs):
    proba = min(1.0, epoch / (n_epochs - 10))
    sampler.sampling_probability.assign(proba)

sampling_probability_cb = keras.callbacks.LambdaCallback(
    on_epoch_begin=update_sampling_probability)
history = model.fit([X_train, X_train_decoder], Y_train, epochs=n_epochs,
                    validation_data=([X_valid, X_valid_decoder], Y_valid),
                    callbacks=[sampling_probability_cb])
```

Not quite 100% validation accuracy, but close enough!

For inference, we could do the exact same thing as earlier, using a `GreedyEmbeddingSampler`. However, just for the sake of completeness, let's use a `SampleEmbeddingSampler` instead. It's almost the same thing, except that instead of using the argmax of the model's output to find the token ID, it treats the outputs as logits and uses them to sample a token ID randomly. This can be useful when you want to generate text. The `softmax_temperature` argument serves the same purpose as when we generated Shakespeare-like text (the higher this argument, the more random the generated text will be).

[ ]:
```
softmax_temperature = tf.Variable(1.)

inference_sampler = tfa.seq2seq.sampler.SampleEmbeddingSampler(
    embedding_fn=decoder_embedding_layer,
    softmax_temperature=softmax_temperature)
inference_decoder = tfa.seq2seq.basic_decoder.BasicDecoder(
    decoder_cell, inference_sampler, output_layer=output_layer,
    maximum_iterations=max_output_length)
batch_size = tf.shape(encoder_inputs)[:1]
start_tokens = tf.fill(dims=batch_size, value=sos_id)
final_outputs, final_state, final_sequence_lengths = inference_decoder(
    start_tokens,
    initial_state=encoder_state,
```

```
    start_tokens=start_tokens,
    end_token=0)

inference_model = keras.models.Model(inputs=[encoder_inputs],
                                     outputs=[final_outputs.sample_id])
```

[ ]:
```
def creative_predict_date_strs(date_strs, temperature=1.0):
    softmax_temperature.assign(temperature)
    X = prepare_date_strs_padded(date_strs)
    Y_pred = inference_model.predict(X)
    return ids_to_date_strs(Y_pred)
```

[ ]:
```
tf.random.set_seed(42)

creative_predict_date_strs(["July 14, 1789", "May 01, 2020"])
```

Dates look good at room temperature. Now let's heat things up a bit:

[ ]:
```
tf.random.set_seed(42)

creative_predict_date_strs(["July 14, 1789", "May 01, 2020"],
                           temperature=5.)
```

Oops, the dates are overcooked, now. Let's call them "creative" dates.

5.3.5 Fifth version: using TFA seq2seq, the Keras subclassing API and attention mechanisms

The sequences in this problem are pretty short, but if we wanted to tackle longer sequences, we would probably have to use attention mechanisms. While it's possible to code our own implementation, it's simpler and more efficient to use TF-Addons's implementation instead. Let's do that now, this time using Keras' subclassing API.

Warning: due to a TensorFlow bug (see this issue for details), the `get_initial_state()` method fails in eager mode, so for now we have to use the subclassing API, as Keras automatically calls `tf.function()` on the `call()` method (so it runs in graph mode).

In this implementation, we've reverted back to using the `TrainingSampler`, for simplicity (but you can easily tweak it to use a `ScheduledEmbeddingTrainingSampler` instead). We also use a `GreedyEmbeddingSampler` during inference, so this class is pretty easy to use:

[ ]:
```
class DateTranslation(keras.models.Model):
    def __init__(self, units=128, encoder_embedding_size=32,
                 decoder_embedding_size=32, **kwargs):
        super().__init__(**kwargs)
        self.encoder_embedding = keras.layers.Embedding(
            input_dim=len(INPUT_CHARS) + 1,
            output_dim=encoder_embedding_size)
        self.encoder = keras.layers.LSTM(units,
```

```
                                        return_sequences=True,
                                        return_state=True)
        self.decoder_embedding = keras.layers.Embedding(
            input_dim=len(OUTPUT_CHARS) + 2,
            output_dim=decoder_embedding_size)
        self.attention = tfa.seq2seq.LuongAttention(units)
        decoder_inner_cell = keras.layers.LSTMCell(units)
        self.decoder_cell = tfa.seq2seq.AttentionWrapper(
            cell=decoder_inner_cell,
            attention_mechanism=self.attention)
        output_layer = keras.layers.Dense(len(OUTPUT_CHARS) + 1)
        self.decoder = tfa.seq2seq.BasicDecoder(
            cell=self.decoder_cell,
            sampler=tfa.seq2seq.sampler.TrainingSampler(),
            output_layer=output_layer)
        self.inference_decoder = tfa.seq2seq.BasicDecoder(
            cell=self.decoder_cell,
            sampler=tfa.seq2seq.sampler.GreedyEmbeddingSampler(
                embedding_fn=self.decoder_embedding),
            output_layer=output_layer,
            maximum_iterations=max_output_length)

    def call(self, inputs, training=None):
        encoder_input, decoder_input = inputs
        encoder_embeddings = self.encoder_embedding(encoder_input)
        encoder_outputs, encoder_state_h, encoder_state_c = self.encoder(
            encoder_embeddings,
            training=training)
        encoder_state = [encoder_state_h, encoder_state_c]

        self.attention(encoder_outputs,
                       setup_memory=True)

        decoder_embeddings = self.decoder_embedding(decoder_input)

        decoder_initial_state = self.decoder_cell.get_initial_state(
            decoder_embeddings)
        decoder_initial_state = decoder_initial_state.clone(
            cell_state=encoder_state)

        if training:
            decoder_outputs, _, _ = self.decoder(
                decoder_embeddings,
                initial_state=decoder_initial_state,
                training=training)
        else:
            start_tokens = tf.zeros_like(encoder_input[:, 0]) + sos_id
```

```
            decoder_outputs, _, _ = self.inference_decoder(
                decoder_embeddings,
                initial_state=decoder_initial_state,
                start_tokens=start_tokens,
                end_token=0)

        return tf.nn.softmax(decoder_outputs.rnn_output)
```

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

model = DateTranslation()
optimizer = keras.optimizers.Nadam()
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])
history = model.fit([X_train, X_train_decoder], Y_train, epochs=25,
                    validation_data=([X_valid, X_valid_decoder], Y_valid))
```

Not quite 100% validation accuracy, but close. It took a bit longer to converge this time, but there were also more parameters and more computations per iteration. And we did not use a scheduled sampler.

To use the model, we can write yet another little function:

[ ]:
```
def fast_predict_date_strs_v2(date_strs):
    X = prepare_date_strs_padded(date_strs)
    X_decoder = tf.zeros(shape=(len(X), max_output_length), dtype=tf.int32)
    Y_probas = model.predict([X, X_decoder])
    Y_pred = tf.argmax(Y_probas, axis=-1)
    return ids_to_date_strs(Y_pred)
```

[ ]:
```
fast_predict_date_strs_v2(["July 14, 1789", "May 01, 2020"])
```

There are still a few interesting features from TF-Addons that you may want to look at: * Using a `BeamSearchDecoder` rather than a `BasicDecoder` for inference. Instead of outputing the character with the highest probability, this decoder keeps track of the several candidates, and keeps only the most likely sequences of candidates (see chapter 16 in the book for more details). * Setting masks or specifying `sequence_length` if the input or target sequences may have very different lengths. * Using a `ScheduledOutputTrainingSampler`, which gives you more flexibility than the `ScheduledEmbeddingTrainingSampler` to decide how to feed the output at time t to the cell at time $t+1$. By default it feeds the outputs directly to cell, without computing the argmax ID and passing it through an embedding layer. Alternatively, you specify a `next_inputs_fn` function that will be used to convert the cell outputs to inputs at the next step.

5.4 10.

Exercise: Go through TensorFlow's Neural Machine Translation with Attention tutorial.

Simply open the Colab and follow its instructions. Alternatively, if you want a simpler example

of using TF-Addons's seq2seq implementation for Neural Machine Translation (NMT), look at the solution to the previous question. The last model implementation will give you a simpler example of using TF-Addons to build an NMT model using attention mechanisms.

5.5 11.

Exercise: Use one of the recent language models (e.g., GPT) to generate more convincing Shakespearean text.

The simplest way to use recent language models is to use the excellent transformers library, open sourced by Hugging Face. It provides many modern neural net architectures (including BERT, GPT-2, RoBERTa, XLM, DistilBert, XLNet and more) for Natural Language Processing (NLP), including many pretrained models. It relies on either TensorFlow or PyTorch. Best of all: it's amazingly simple to use.

First, let's load a pretrained model. In this example, we will use OpenAI's GPT model, with an additional Language Model on top (just a linear layer with weights tied to the input embeddings). Let's import it and load the pretrained weights (this will download about 445MB of data to `~/.cache/torch/transformers`):

[ ]:
```
from transformers import TFOpenAIGPTLMHeadModel

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

Next we will need a specialized tokenizer for this model. This one will try to use the spaCy and ftfy libraries if they are installed, or else it will fall back to BERT's `BasicTokenizer` followed by Byte-Pair Encoding (which should be fine for most use cases).

[ ]:
```
from transformers import OpenAIGPTTokenizer

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

Now let's use the tokenizer to tokenize and encode the prompt text:

[ ]:
```
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
```

Easy! Next, let's use the model to generate text after the prompt. We will generate 5 different sentences, each starting with the prompt text, followed by 40 additional tokens. For an explanation of what all the hyperparameters do, make sure to check out this great blog post by Patrick von Platen (from Hugging Face). You can play around with the hyperparameters to try to obtain better results.

[ ]:
```
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
```

Now let's decode the generated sequences and print them:

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

You can try more recent (and larger) models, such as GPT-2, CTRL, Transformer-XL or XLNet, which are all available as pretrained models in the transformers library, including variants with Language Models on top. The preprocessing steps vary slightly between models, so make sure to check out this generation example from the transformers documentation (this example uses PyTorch, but it will work with very little tweaks, such as adding `TF` at the beginning of the model class name, removing the `.to()` method calls, and using `return_tensors="tf"` instead of `"pt"`.

Hope you enjoyed this chapter! :)