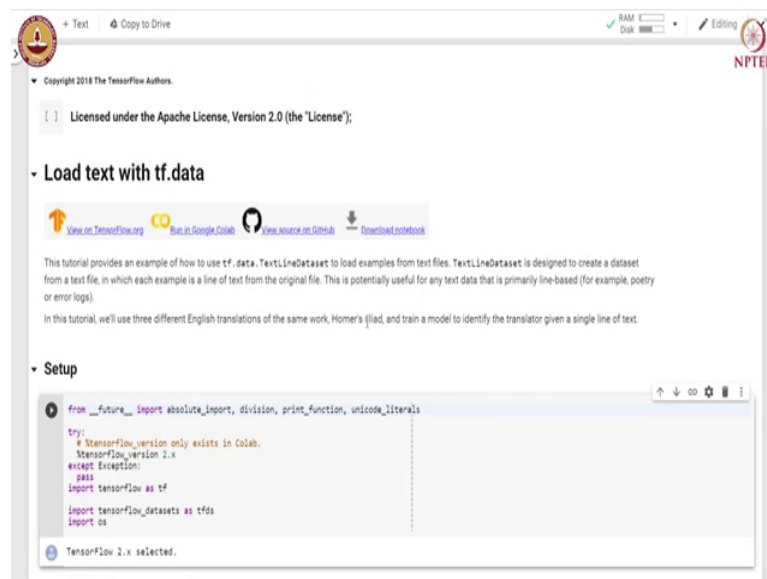


Practical Machine Learning
Dr. Ashish Tendulkar
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay

Lecture – 14
Building Data Pipelines for TensorFlow – Part 2

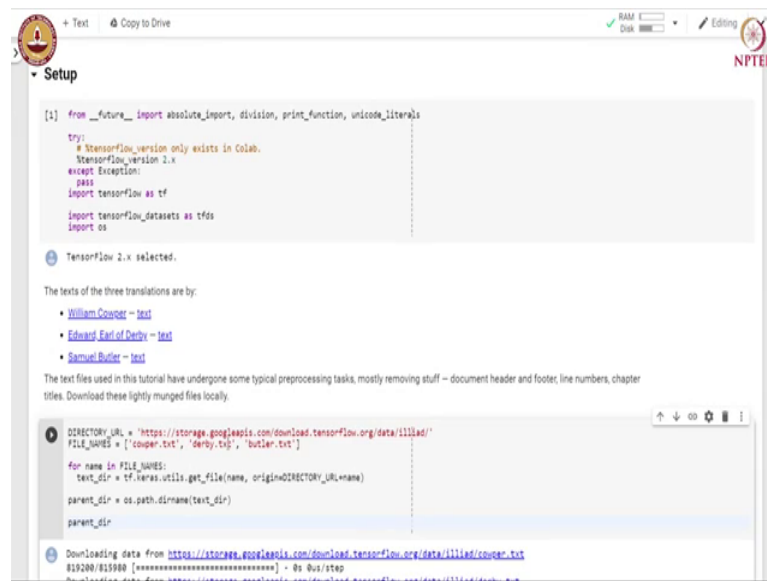
So, let us look at how to load the text data and create in pipelines based on the text data.

(Refer Slide Time: 00:22)



So, in this case we use `tf.data.textLineDataset` to load examples from text file into a dataset. The text line data set is designed to create a data set from a text file in which each example is a line of text from the original file. This is very important to note that each example is a line of text from the input file. This is potentially useful for any text data that is primarily line based. This kind of data sets could be a poetry, error logs, movie reviews etc.

(Refer Slide Time: 01:14)



```
[1] from __future__ import absolute_import, division, print_function, unicode_literals

try:
    # TensorFlow version only exists in Colab.
    TensorFlow_version 2.x
except Exception:
    pass
import tensorflow as tf
import tensorflow_datasets as tfds
import os

TensorFlow 2.x selected.

The texts of the three translations are by:
• William Cowper - text
• Edward Earl of Derby - text
• Samuel Butler - text

The text files used in this tutorial have undergone some typical preprocessing tasks, mostly removing stuff – document header and footer, line numbers, chapter titles. Download these lightly munged files locally.

DIRECTOR_URL = 'https://storage.googleapis.com/download.tensorflow.org/data/iliad/'
FILE_NAMES = ['cowper.txt', 'derby.txt', 'butler.txt']

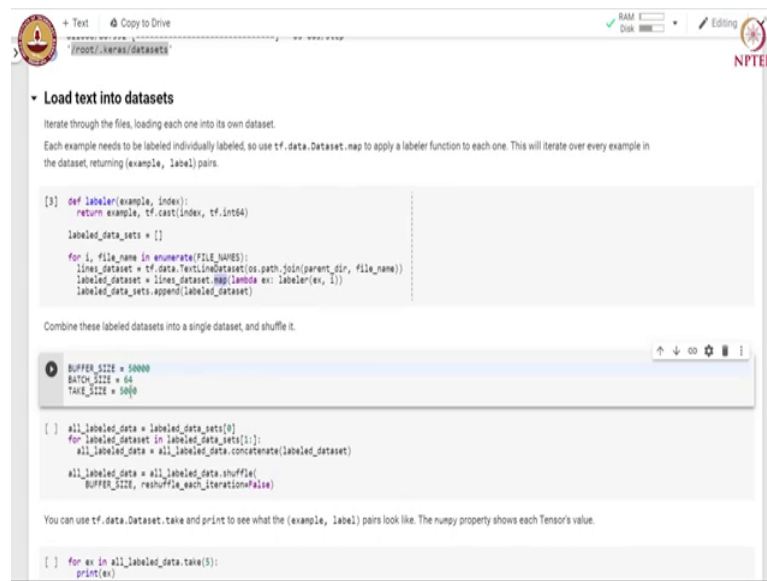
for name in FILE_NAMES:
    text_dir = tf.keras.utils.get_file(name, origin=DIRECTOR_URL+name)
    parent_dir = os.path.dirname(text_dir)
    parent_dir

Downloading data from https://storage.googleapis.com/download.tensorflow.org/data/iliad/cowper.txt
819200/819200 [*****] - 0s 0us/step
Downloading data from https://storage.googleapis.com/download.tensorflow.org/data/iliad/derby.txt
```

So, let us start by importing TensorFlow and TensorFlow data set. We will be using 3 different in English translations of the same work that is Horner's Iliad and see how to use text line data set and other pre processing on the text to create a data set that can be fade into the model for training. So, we have text of 3 translations as input. So, there are 3 files cowper.txt, derby.txt and butler.txt which is translation of Homer Iliad.

We first download these files and then print the parent directory. After getting the data the normal pre-processing tasks are about removing document header and footer, line numbers, chapter titles if those are present in the file. So, you can see that now the files are downloaded and they are in root/.keras/datasets directory.

(Refer Slide Time: 02:45)



The screenshot shows a Jupyter Notebook with the following content:

Load text into datasets

Iterate through the files, loading each one into its own dataset.

Each example needs to be labeled individually labeled, so use `tf.data.Dataset.map` to apply a `labeler` function to each one. This will iterate over every example in the dataset, returning (example, label) pairs.

```
[3] def labeler(example, index):  
    return example, tf.cast(index, tf.int64)  
  
    labeled_data_sets = []  
  
    for i, file_name in enumerate(FILE_NAMES):  
        lines_dataset = tf.data.TextLineDataset(os.path.join(parent_dir, file_name))  
        labeled_dataset = lines_dataset.map(lambda ex, i: labeler(ex, i))  
        labeled_data_sets.append(labeled_dataset)
```

Combine these labeled datasets into a single dataset, and shuffle it.

```
BUFFER_SIZE = 50000  
BATCH_SIZE = 64  
TAKE_SIZE = 5000
```

```
[ ] all_labeled_data = labeled_data_sets[0]  
    for labeled_dataset in labeled_data_sets[1:]:  
        all_labeled_data = all_labeled_data.concatenate(labeled_dataset)  
  
    all_labeled_data = all_labeled_data.shuffle(  
        BUFFER_SIZE, reshuffle_each_iteration=False)
```

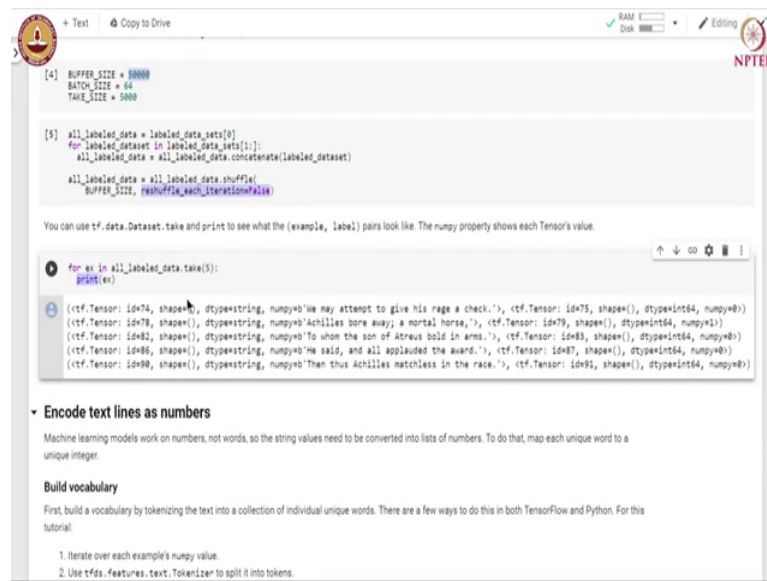
You can use `tf.data.Dataset.take` and `print` to see what the (example, label) pairs look like. The `numpy` property shows each `Tensor`'s value.

```
[ ] for ex in all_labeled_data.take(5):  
    print(ex)
```

Now, that we have downloaded the files we will go through each of the file. We will load them into a dataset each file will be a single dataset, we need to label each example for that purpose we use `dataset.map()` function that applies a `labeler()` function on each example. So, `labeler()` function takes an example and assign a label to that particular example and we do that with a `map()` function.

So, `map` will iterate over every examples in the dataset and we will return example comma label pairs. So, let us apply the labels on each of the example by running this particular code cell. We will combine this label dataset into a single dataset and shuffle it for that we set the buffer size to 50,000. We are going to use batch size of 64 and we will look at 5,000 examples.

(Refer Slide Time: 04:05)



```
[4] BUFFER_SIZE = 50000
    BATCH_SIZE = 64
    TAKE_SIZE = 5000

[5] all_labeled_data = labeled_data_sets[0]
    for labeled_dataset in labeled_data_sets[1:]:
        all_labeled_data = all_labeled_data.concatenate(labeled_dataset)
    all_labeled_data = all_labeled_data.shuffle(
        BUFFER_SIZE, reshuffle_each_iteration=False)

You can use tf.data.Dataset.take and print to see what the (example, label) pairs look like. The numpy property shows each Tensor's value.

for ex in all_labeled_data.take(5):
    print(ex)

(tf.Tensor: id=74, shape=(), dtype=string, numpy='We may attempt to give his rage a check.'). (tf.Tensor: id=75, shape=(), dtype=int64, numpy=0)
(tf.Tensor: id=76, shape=(), dtype=string, numpy='Achilles bore away a mortal horse,'). (tf.Tensor: id=77, shape=(), dtype=int64, numpy=1)
(tf.Tensor: id=78, shape=(), dtype=string, numpy='To whom the son of Atreus bold in arms.'). (tf.Tensor: id=79, shape=(), dtype=int64, numpy=0)
(tf.Tensor: id=80, shape=(), dtype=string, numpy='He said, and all applauded the award.'). (tf.Tensor: id=81, shape=(), dtype=int64, numpy=0)
(tf.Tensor: id=82, shape=(), dtype=string, numpy='Then thus Achilles matchless in the race.'). (tf.Tensor: id=83, shape=(), dtype=int64, numpy=0)
```

Encode text lines as numbers

Machine learning models work on numbers, not words, so the string values need to be converted into lists of numbers. To do that, map each unique word to a unique integer.

Build vocabulary

First, build a vocabulary by tokenizing the text into a collection of individual unique words. There are a few ways to do this in both TensorFlow and Python. For this tutorial:

1. Iterate over each example's numpy value.
2. Use `tfds.features.text.Tokenizer` to split it into tokens.

So, let us concatenate the individual datasets into a single dataset. You can see that first we copy the first dataset into all label data and then for the remaining datasets we combine we over remaining datasets and concatenate them to the all label data. So, at the end of this particular process all the datasets are concatenated into a single dataset which is all label data.

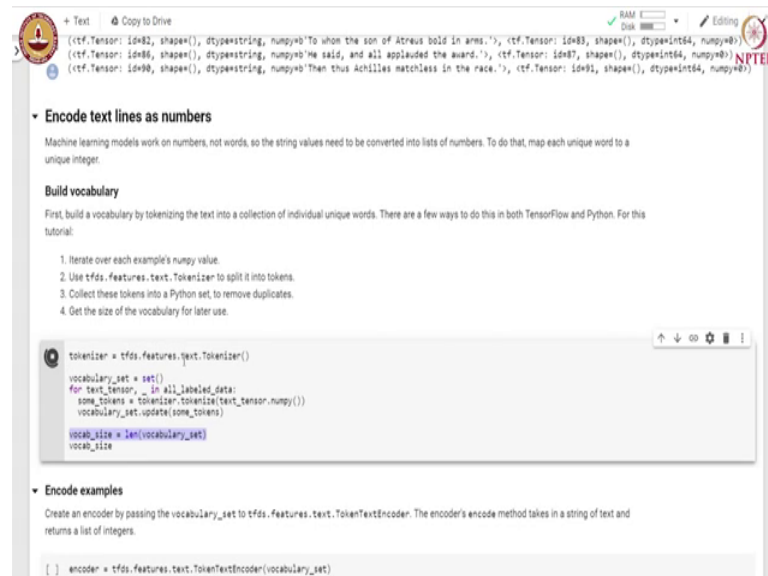
After concatenating all datasets next we shuffle the datasets using the buffer size of 50,000 and we set the *reshuffled_each_iteration* parameters to *false*. So, we do not want to reshuffle the dataset at each iteration then we can use `take()` and `print()` functions to see what example pair look like. So, what we do is we take 5 examples from all label data and we print each of the example.

We have 5 tensors for first 5 example, one per each example. We can see that each tensor is a scalar quantity or a 0 dimensional tensor. It is of type string and we see the string corresponding to this particular tensor. Corresponding to each of the sentence there is a label associated with it which is another scalar and a label for the first sentence is 0.

Second sentence has got label of one and so on. So, you can see the first 5 examples or first 5 examples from `all_labeled_data`. Now you know that machine learning models

work on numbers and not words. So, naturally we need to convert these words into numbers.

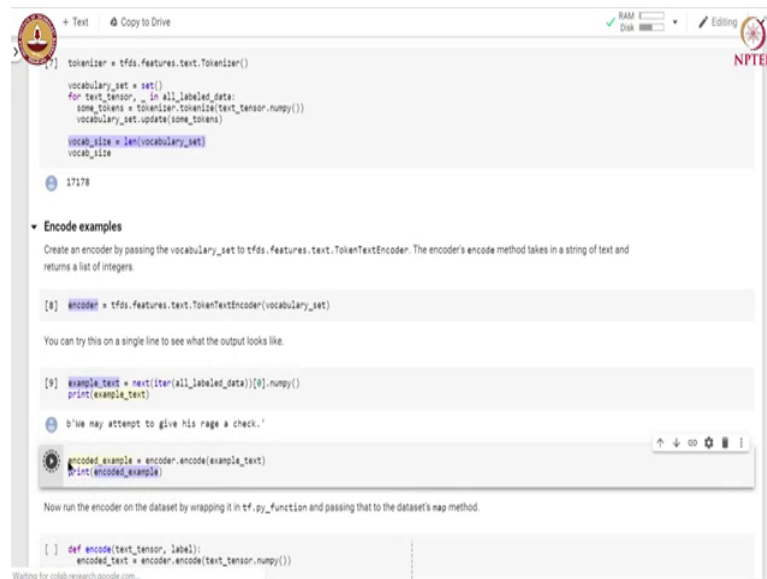
(Refer Slide Time: 06:58)



We can map each unique word to unique integer. So, first you build a vocabulary by tokenizing the text into collection of individual words. There are few ways to do this both in TensorFlow and Python. Here we will iterate over each example and we will tokenize each example into tokens. We then collect each of the tokens and remove the duplicate.

So, we can note that the vocabulary set is a set that we update with the tokens that we get after tokenization. So, the effect of that is the duplicate tokens are removed. Finally, after completing the process for each sentence we can find out how many words are there in the vocabulary just by taking the length of the vocabulary set.

(Refer Slide Time: 08:37)



```
tokenizer = tfds.features.text.TokenTextEncoder()

vocabulary_set = set()
for text_tensor, _ in all_labeled_data:
    some_tokens = tokenizer.tokenize(text_tensor.numpy())
    vocabulary_set.update(some_tokens)

vocab_size = len(vocabulary_set)
vocab_size
```

17178

Encode examples

Create an encoder by passing the vocabulary_set to tfds.features.text.TokenTextEncoder. The encoder's encode method takes in a string of text and returns a list of integers.

```
[8] encoder = tfds.features.text.TokenTextEncoder(vocabulary_set)
```

You can try this on a single line to see what the output looks like.

```
[9] example_text = next(iter(all_labeled_data))[0].numpy()
    print(example_text)
```

b'we may attempt to give his rage a check.'

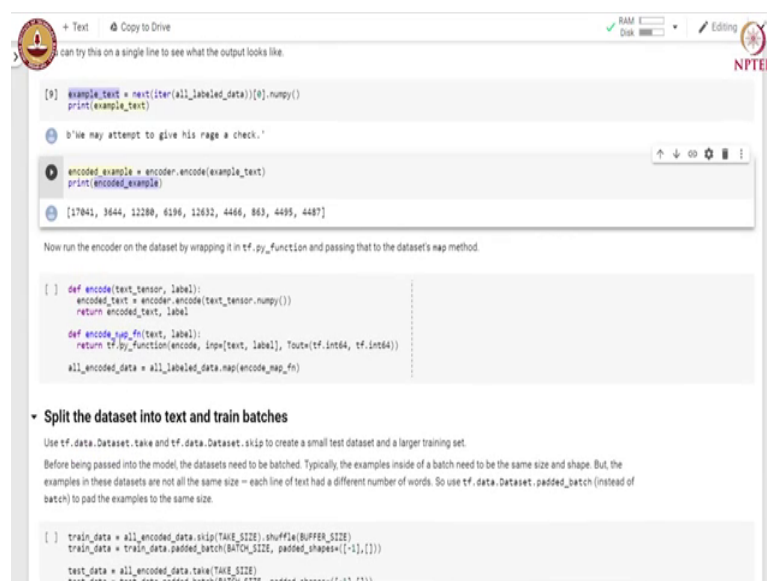
```
encoded_example = encoder.encode(example_text)
print(encoded_example)
```

Now run the encoder on the dataset by wrapping it in tf.py_function and passing that to the dataset's map method.

```
[ ] def encode(text_tensor, label):
    encoded_text = encoder.encode(text_tensor.numpy())
    return encoded_text, label
```

So, we have 17,178 words in our vocabulary. We create an encoder by passing the vocabulary set to token text encoder which takes a string of text. The token text encoder has encode() method that takes a string of text and returns a list of integers. We take the example text and we encode that example text with the encoder defined over here and then we print the encoded example. So, we can see that the first text here is we may attempt to give his rage a check.

(Refer Slide Time: 09:36)



```
[9] example_text = next(iter(all_labeled_data))[0].numpy()
    print(example_text)
```

b'we may attempt to give his rage a check.'

```
encoded_example = encoder.encode(example_text)
print(encoded_example)
```

[17041, 3644, 12280, 6186, 12632, 4466, 863, 4495, 4487]

Now run the encoder on the dataset by wrapping it in tf.py_function and passing that to the dataset's map method.

```
[ ] def encode(text_tensor, label):
    encoded_text = encoder.encode(text_tensor.numpy())
    return encoded_text, label

def encode_map_fn(text, label):
    return tf.py_function(encode, [text, label], Tout=(tf.int64, tf.int64))

all_encoded_data = all_labeled_data.map(encode_map_fn)
```

Split the dataset into text and train batches

Use tf.data.Dataset.take and tf.data.Dataset.skip to create a small test dataset and a larger training set.

Before being passed into the model, the datasets need to be batched. Typically, the examples inside of a batch need to be the same size and shape. But, the examples in these datasets are not all the same size – each line of text had a different number of words. So use tf.data.Dataset.padded_batch (instead of batch) to pad the examples to the same size.

```
[ ] train_data = all_encoded_data.skip(TAKE_SIZE).shuffle(BUFFER_SIZE)
    train_data = train_data.padded_batch(BATCH_SIZE, padded_shapes=[-1, []])

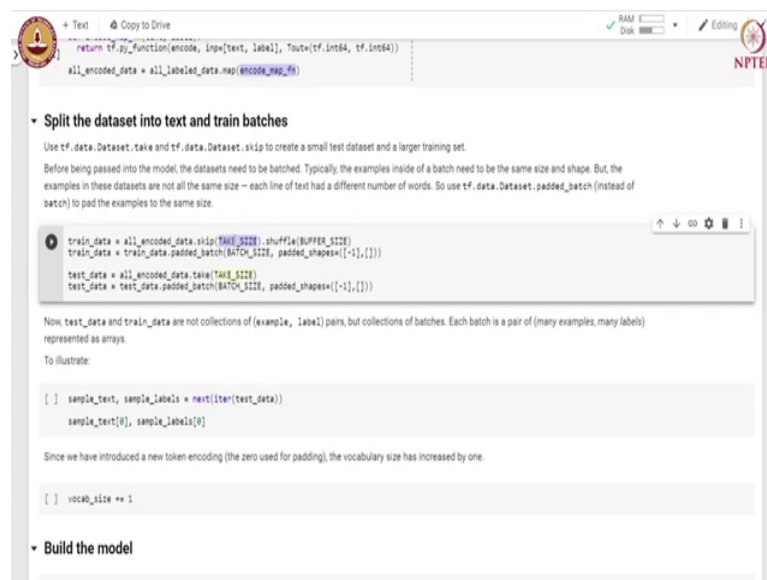
    test_data = all_encoded_data.take(TAKE_SIZE)
    test_data = test_data.padded_batch(BATCH_SIZE, padded_shapes=[-1, []])
```

Let us see how this example is encoded. Each of the word over here is encoded with a unique integer. Now, that we have seen how the encoder works on a single example we

will apply the encoder on each example in the dataset. For that we wrap the encode under `tf.py_function()`.

The `encode()` function takes a text tensor and label and returns the encoded text along with its label. So, we use `tf.py_function()` to wrap encode and we give the text and labels as inputs. So, we use a `map()` function to apply the `encode_function_map` on each example. So, that each line gets converted into its numeric representation.

(Refer Slide Time: 11:03)



The next task is to split the dataset into test and training batches. We use `tf.data.dataset.take` and `tf.data.dataset.skip` to create a small test set and a large training set. Before being passed into the model we need to batch the dataset. Typically the examples inside a batch need to be of the same size and shape. But the examples in these datasets are not all of the same size. Remember each line of text had a different number of words. So, we use `padded_batch` instead of `batch` to pad each example to make it of the same size. So, let us see how to do that. So, let us first conceive how to construct the training data.

So, we take all encoded data and we skip first 5,000 examples because we have taken `take_size` to be 5,000. Then we shuffle these examples. So, by skipping the first 5,000

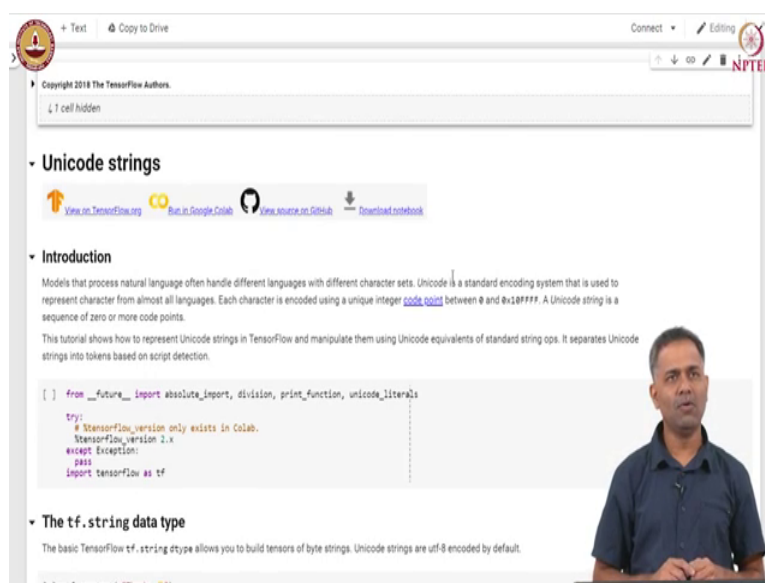
examples and shuffling we get the training data and we take these 5,000 examples that we skipped in the training data to form the test data. Then for training and test data we apply a padded batch so that each example is made of the same size by padding.

So, let us run this code cell now we have test data and training data ready. So, `test_data` and `train_data` are not collections of (examples, labels), but they are collection of batches and each batch is a pair of many (examples, labels) represented as arrays.

Let us look at the first batch. We use an iterate over the test data and we take the next. So, that this next returns the text and labels which we store in `sample_text` and `sample_labels` and then we examine them. So, you can see that the first example is already converted into numbers and it is padded with 0 so that its length is equal to the length of the longest sequence in the dataset and we also have a label which is 0 in this particular case.

Since we have introduced a new token that is 0 for padding we increase the size of vocabulary by one. So, these are steps that we take in order to convert the text data into dataset. We use text line data set method to construct data set object from the text data. Now that we have constructed training and test data we can use that for building for training the model.

(Refer Slide Time: 15:38)



Copyright 2018 The TensorFlow Authors.

1 cell hidden

Unicode strings

[View on TensorFlow.org](#) [Run in Google Colab](#) [View source on GitHub](#) [Download notebook](#)

Introduction

Models that process natural language often handle different languages with different character sets. Unicode is a standard encoding system that is used to represent character from almost all languages. Each character is encoded using a unique integer `code point` between 0 and 0x10FFFF. A Unicode string is a sequence of zero or more code points.

This tutorial shows how to represent Unicode strings in TensorFlow and manipulate them using Unicode equivalents of standard string ops. It separates Unicode strings into tokens based on script detection.

```
[ ] from __future__ import absolute_import, division, print_function, unicode_literals

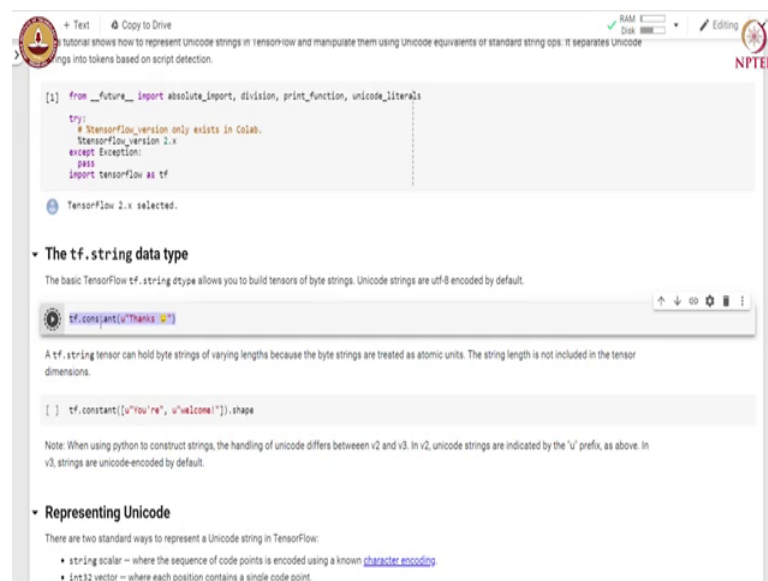
try:
    # TensorFlow version only exists in Colab.
    TensorFlow_version 2.x
except Exception:
    pass
import tensorflow as tf
```

The tf.string data type

The basic TensorFlow `tf.string` dtype allows you to build tensors of byte strings. Unicode strings are utf-8 encoded by default.

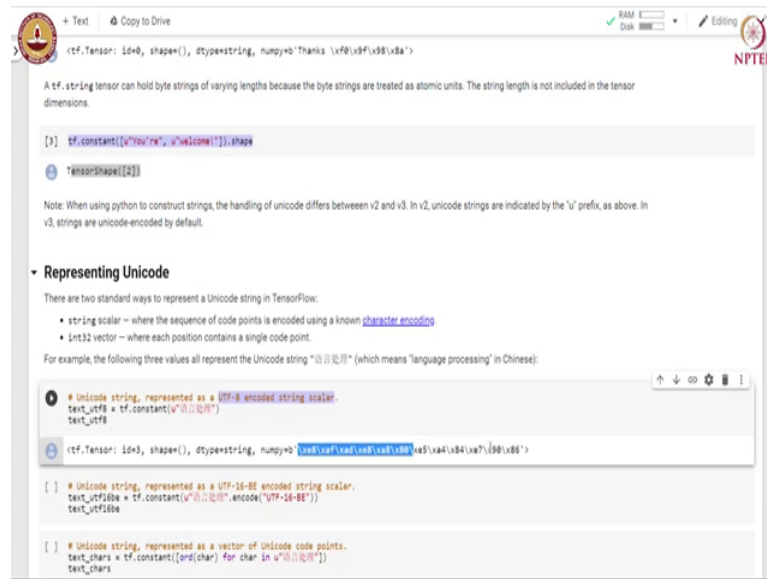
Let us understand how we can process unicode strings with TensorFlow and also understand how to construct data set objects with unicode strings. So, we can use TensorFlow to process different languages with different character sets. Unicode is a standard encoding system that is used to represent characters from almost all languages. Each character is encoded using a unique integer code point between 0 and this particular number. A unicode string is a sequence of 0 or more code points.

(Refer Slide Time: 16:46)



So, the *tf.stringdtype* allows us to build tensor of byte strings. Unicode strings are UTF-8 encoded by default. So, let us see how this thanks with emoji is represented. We can define that with *tf.constant*.

(Refer Slide Time: 17:19)



The screenshot shows a Jupyter Notebook interface with a code cell containing the following Python code:

```
<tf.Tensor: id=0, shape=(), dtype=string, numpy=b'Thanks \xf0\x9f\x98\xba'>
```

Below the code, there is a text box explaining that a `tf.string` tensor holds byte strings of varying lengths, treated as atomic units, and that the string length is not included in the tensor dimensions.

The code cell also shows the output of `tf.constant(["You're", "welcome"]).shape`, which is `tensorshape([2])`.

A note states: "When using python to construct strings, the handling of unicode differs between v2 and v3. In v2, unicode strings are indicated by the 'u' prefix, as above. In v3, strings are unicode-encoded by default."

The notebook has a section titled "Representing Unicode" with the following text:

There are two standard ways to represent a Unicode string in TensorFlow:

- string scalar – where the sequence of code points is encoded using a known [character encoding](#)
- int32 vector – where each position contains a single code point.

For example, the following three values all represent the Unicode string "语言处理" (which means "language processing" in Chinese):

```
# Unicode string, represented as a UTF-8 encoded string scalar.
text_utf8 = tf.constant(u"语言处理")
text_utf8

# Unicode string, represented as a UTF-16-BE encoded string scalar.
text_utf16be = tf.constant(u"语言处理".encode("UTF-16-BE"))
text_utf16be

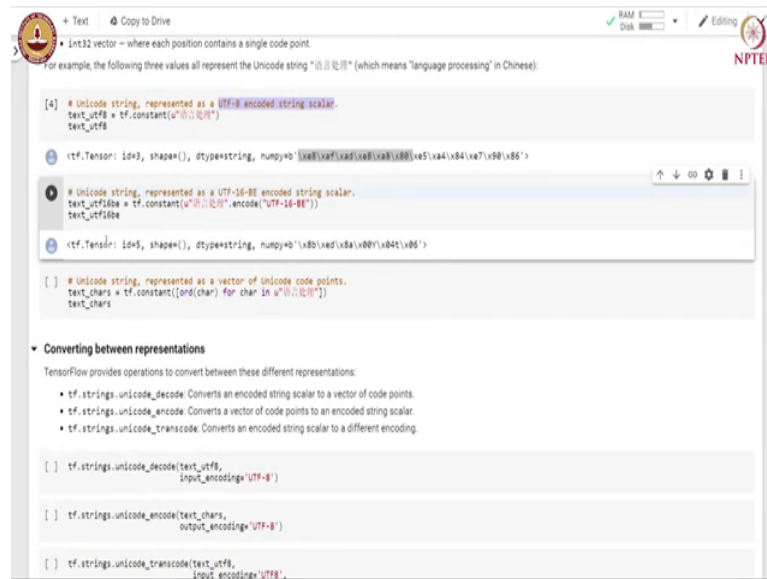
# Unicode string, represented as a vector of Unicode code points.
text_chars = tf.constant([ord(char) for char in u"语言处理"])
text_chars
```

A *tf.string* tensor can hold byte strings of varying length because the byte strings are treated as atomic units. The string length is not included in the tensor dimension. Let us look at the shape of these 2 strings which has “You are” and “welcome” as two strings. So, you can see that this is a tensor which is a vector containing 2 elements or in other words this is a 1d tensor with shape 2 there are two ways to represent a unicode string in TensorFlow.

One is using a string scalar or integer 32 vector. The string scalar stores the sequence of code points with a known character encoding, whereas integer 32 vector stores each position containing a single code point.

So, as an example the following 3 values are represented in a unicode string. This is a Chinese character for language processing. Here, we are using unicode strings represented as a UTF-8 encoded string scalar. So, you can see that this is a scalar with dtype string and we have UTF-8 encoded strings string scalars.

(Refer Slide Time: 19:21)



```
+ Text Copy to Drive
• Int32 vector - where each position contains a single code point.
For example, the following three values all represent the Unicode string "语言" (which means "language processing" in Chinese)

[A] # Unicode string, represented as a UTF-8 encoded string scalar.
text_utf8 = tf.constant(u"语言")
text_utf8

<tf.Tensor: id=3, shape=(), dtype=string, numpy=b'\xe8\xaf\xad\xe8\xbf\x84\xe5\x84\x87\xe9\x86'>

[B] # Unicode string, represented as a UTF-16-BE encoded string scalar.
text_utf16be = tf.constant(u"语言".encode("UTF-16-BE"))
text_utf16be

<tf.Tensor: id=4, shape=(), dtype=string, numpy=b'\xb8\xed\xba\x80\x84\xe4\x86'>

[C] # Unicode string, represented as a vector of Unicode code points.
text_chars = tf.constant([ord(char) for char in u"语言"])
text_chars

Converting between representations
TensorFlow provides operations to convert between these different representations:
• tf.strings.unidecode: Converts an encoded string scalar to a vector of code points.
• tf.strings.unidecode_encode: Converts a vector of code points to an encoded string scalar.
• tf.strings.unidecode_transcode: Converts an encoded string scalar to a different encoding.

[ ] tf.strings.unidecode(text_utf8,
    input_encoding='UTF-8')

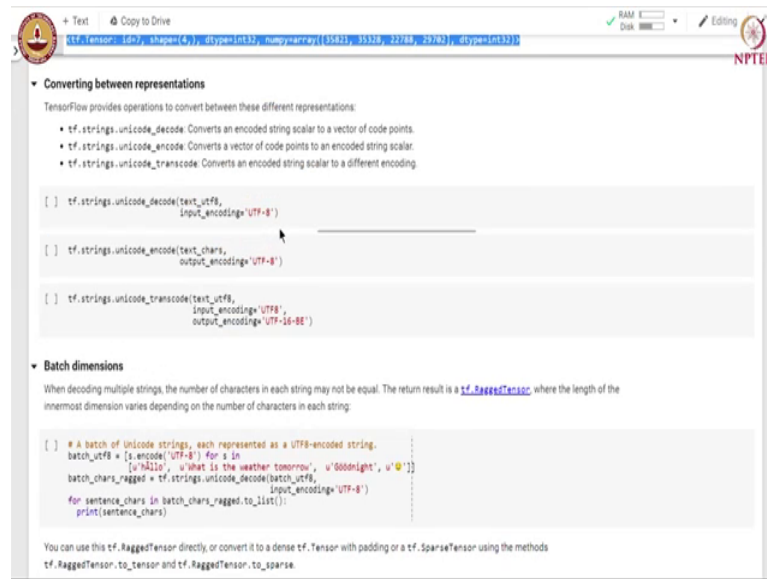
[ ] tf.strings.unidecode_encode(text_chars,
    output_encoding='UTF-8')

[ ] tf.strings.unidecode_transcode(text_utf8,
    input_encoding='UTF8',
```

Here we have a unicode string represented as UTF-16-BE encoded string scalars and finally, we have the same unicode string represented as a vector of unicode code points. So, this text character is a vector containing 4 elements and each element is a unicode code point corresponding to the character.

So, this 35821 is a code point corresponding to this particular character 35328 is the code point corresponding to this character and so on. So, there are 4 characters and there are 4 code points corresponding to each one of them. So, we just saw 3 ways of representing strings one using UTF-8 encoded string scalar then UTF-16-BE encoded string scalars and as a vector of unicode code points.

(Refer Slide Time: 20:38)



The screenshot shows a Jupyter Notebook interface with a code cell. At the top, a status bar indicates 'RAM' and 'Disk' usage. The code cell contains the following text:

```
tf.Tensor('107', shape=(4,), dtype=int32, numpy=array([35821, 35329, 32788, 39782], dtype=int32))
```

Below the code cell, there is a section titled 'Converting between representations'. It states: 'TensorFlow provides operations to convert between these different representations:'

- `tf.strings.unicode_decode`: Converts an encoded string scalar to a vector of code points.
- `tf.strings.unicode_encode`: Converts a vector of code points to an encoded string scalar.
- `tf.strings.unicode_transcode`: Converts an encoded string scalar to a different encoding.

Below the list, there are three code snippets:

```
[ ] tf.strings.unicode_decode(text_utf8,
                             input_encoding='UTF-8')

[ ] tf.strings.unicode_encode(text_chars,
                             output_encoding='UTF-8')

[ ] tf.strings.unicode_transcode(text_utf8,
                                input_encoding='UTF8',
                                output_encoding='UTF-16-BE')
```

Below the code snippets, there is a section titled 'Batch dimensions'. It states: 'When decoding multiple strings, the number of characters in each string may not be equal. The return result is a `tf.RaggedTensor`, where the length of the innermost dimension varies depending on the number of characters in each string.'

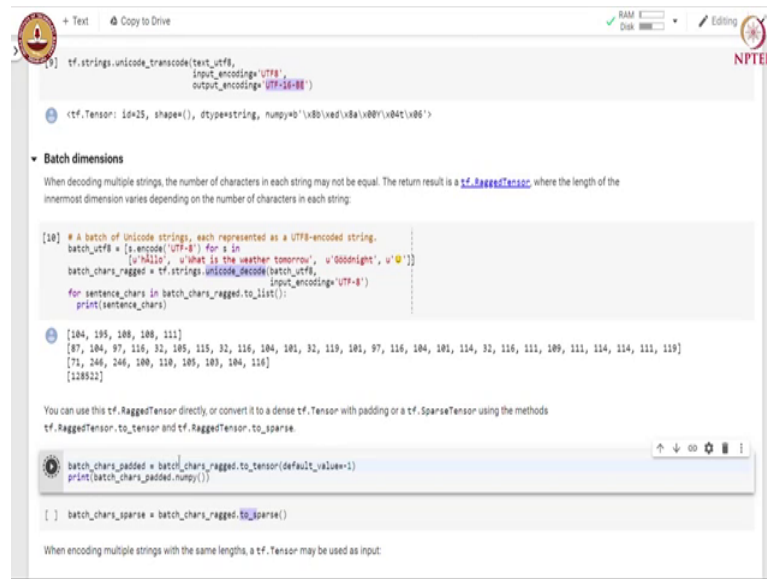
```
[ ] # A batch of Unicode strings, each represented as a UTF8-encoded string.
batch_utf8 = [s.encode('UTF-8') for s in
              ['hello', 'What is the weather tomorrow', 'Goodnight', '']]
batch_chars_ragged = tf.strings.unicode_decode(batch_utf8,
                                              input_encoding='UTF-8')
for sentence_chars in batch_chars_ragged.to_list():
    print(sentence_chars)
```

At the bottom, there is a note: 'You can use this `tf.RaggedTensor` directly, or convert it to a dense `tf.Tensor` with padding or a `tf.SparseTensor` using the methods `tf.RaggedTensor.to_tensor` and `tf.RaggedTensor.to_sparse`.'

TensorFlow provides operations to convert between these different representations. We use `tf.strings.unicode_decode` to convert an encoded string scalar to a vector of code point. `Unicode_encode` converts a vector of code points to an encoded string scalar and `unicode_transcode` converts an encoded string scalar to a different encoding. Let us look at the examples. Here we use `unicode_decode`. This `text_utf8` will get converted to a vector of code points.

Let us take the vector of code points and convert that to a string scalar using `unicode_encode` method. When we apply that the vector of code points get converted into a string scalar encoded in UTF-8 finally, we can use `encode_transcode` to convert between UTF-8 encoding and UTF-16-BE encoding. So, the input encoding is UTF-8 and the output encoding is UTF-16-BE.

(Refer Slide Time: 22:34)



```
[9] tf.strings.unicode_encode(text_utf8,
                             input_encoding='UTF8',
                             output_encoding='UTF-8')

<tf.Tensor: id=25, shape=(), dtype=string, numpy=b'\x0b\xad\xba\x0b\x04t\x06'>
```

Batch dimensions

When decoding multiple strings, the number of characters in each string may not be equal. The return result is a [tf.RaggedTensor](#), where the length of the innermost dimension varies depending on the number of characters in each string:

```
[10] # a batch of unicode strings, each represented as a UTF-8 encoded string.
batch_utf8 = [s.encode('UTF-8') for s in
              ['hello', 'what is the weather tomorrow', 'goodnight', '👋']]
batch_chars_ragged = tf.strings.unicode_decode(batch_utf8,
                                                input_encoding='UTF-8')
for sentence_chars in batch_chars_ragged.to_list():
    print(sentence_chars)

[104, 105, 106, 108, 111]
[87, 104, 97, 116, 32, 105, 115, 32, 116, 104, 101, 32, 119, 101, 97, 116, 104, 101, 114, 32, 116, 111, 109, 111, 114, 111, 119]
[71, 246, 246, 100, 110, 101, 104, 110]
```

You can use this `tf.RaggedTensor` directly or convert it to a dense `tf.Tensor` with padding or a `tf.SparseTensor` using the methods `tf.RaggedTensor.to_tensor` and `tf.RaggedTensor.to_sparse`.

```
batch_chars_padded = batch_chars_ragged.to_tensor(default_value=1)
print(batch_chars_padded.numpy())

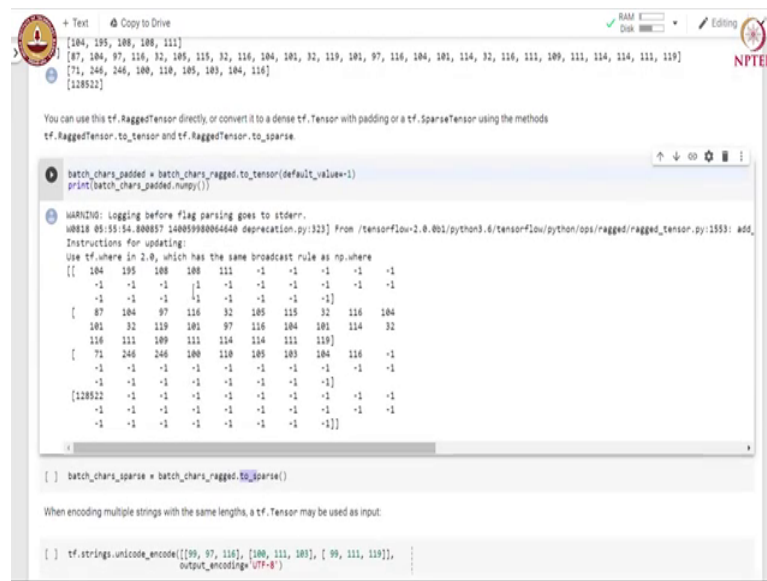
[ ] batch_chars_sparse = batch_chars_ragged.to_sparse()
```

When encoding multiple strings with the same lengths, a `tf.Tensor` may be used as input:

When decoding multiple strings the number of characters in each string may not be equal. So, it returns `tf.RaggedTensor` where the length of inner most dimension varies depending on the number of characters in each string. We take a batch of unicode strings each represented as a UTF-8 encoded string. So, we have hello what is the weather tomorrow then goodnight and an emoji for smiley.

Use `encode string()` function that returns we use `unicode_decode()` function to convert the characters into a string of unicode code points. So, we get a `RaggedTensor` and you can see that the length of each vector resulting vector is different. We can use this `RaggedTensor` directly or convert that into `tf.tensor` with padding or to `tf.sparse tensor` using `to_sparse()` method.

(Refer Slide Time: 24:21)



A Jupyter Notebook interface showing a code cell with the following content:

```
batch_chars_padded = batch_chars_ragged.to_tensor(default_value=-1)
print(batch_chars_padded.numpy())
```

The output shows a 4x12 matrix of integers, where the first three rows represent padded character indices and the fourth row is a constant -1. A warning message is displayed above the output:

```
WARNING: Logging before flag parsing goes to stderr.
10818 05:55:54.800857 14085998086400 deprecation.py:323] From /tensorflow/2.0.0/python3.6/tensorflow/python/ops/ragged/ragged_tensor.py:1553: add,
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
[[ 104 195 108 108 111 -1 -1 -1 -1 -1 -1 -1
  -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
  -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
 [ 87 104 97 116 32 105 115 32 116 104
  101 32 119 101 97 116 104 101 114 32
  116 111 109 111 114 114 111 119]
 [ 71 246 246 100 110 105 103 104 116 -1
  -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
  -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 [128522 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
  -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
  -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
```

Below the code cell, there is a text box with the following text:

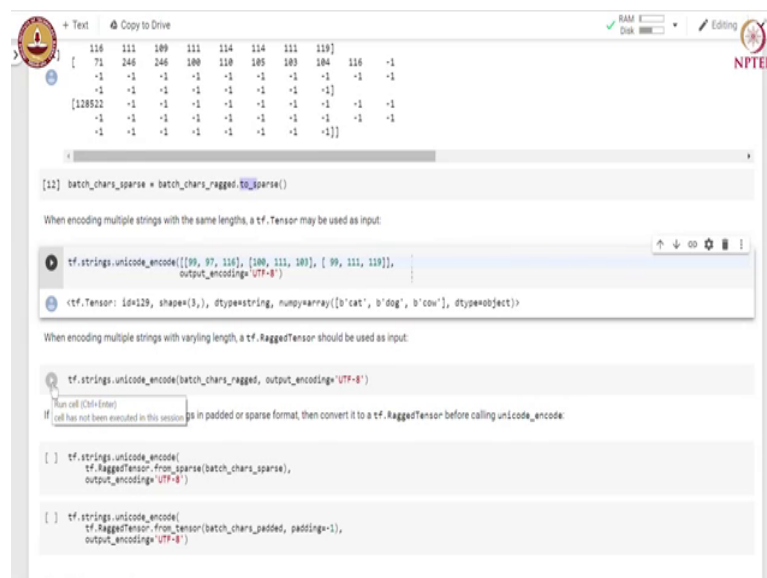
You can use this `tf.RaggedTensor` directly, or convert it to a dense `tf.Tensor` with padding or a `tf.SparseTensor` using the methods `tf.RaggedTensor.to_tensor` and `tf.RaggedTensor.to_sparse`.

When encoding multiple strings with the same length, a `tf.Tensor` may be used as input.

```
[ ] tf.strings.unicode_encode([[99, 97, 116], [100, 111, 103], [ 99, 111, 119]],
                             output_encoding='UTF-8')
```

Now, here we converted the RaggedTensor to a dense tf.tensor. Here we pad each example with -1. So, that the lengths of all the examples are the same.

(Refer Slide Time: 24:47)



A Jupyter Notebook interface showing a code cell with the following content:

```
[12] batch_chars_sparse = batch_chars_ragged.to_sparse()
```

The output shows a 4x12 matrix of integers, where the first three rows represent padded character indices and the fourth row is a constant -1. A warning message is displayed above the output:

```
WARNING: Logging before flag parsing goes to stderr.
10818 05:55:54.800857 14085998086400 deprecation.py:323] From /tensorflow/2.0.0/python3.6/tensorflow/python/ops/ragged/ragged_tensor.py:1553: add,
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
[[ 104 195 108 108 111 -1 -1 -1 -1 -1 -1 -1
  -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
  -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
 [ 87 104 97 116 32 105 115 32 116 104
  101 32 119 101 97 116 104 101 114 32
  116 111 109 111 114 114 111 119]
 [ 71 246 246 100 110 105 103 104 116 -1
  -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
  -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 [128522 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
  -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
  -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
```

Below the code cell, there is a text box with the following text:

When encoding multiple strings with the same length, a `tf.Tensor` may be used as input.

```
[ ] tf.strings.unicode_encode([[99, 97, 116], [100, 111, 103], [ 99, 111, 119]],
                             output_encoding='UTF-8')
```

The output shows a `tf.Tensor` object with the following details:

```
<tf.Tensor 'id=129, shape=(3,), dtype=string, numpy=array([b'cat', b'dog', b'cow'], dtype=object)>
```

When encoding multiple strings with varying length, a `tf.RaggedTensor` should be used as input.

```
[ ] tf.strings.unicode_encode(batch_chars_ragged, output_encoding='UTF-8')
```

The output shows a `tf.RaggedTensor` object with the following details:

```
<tf.RaggedTensor 'id=129, shape=(3, 12), dtype=string, numpy=array([b'cat', b'dog', b'cow'], dtype=object)>
```

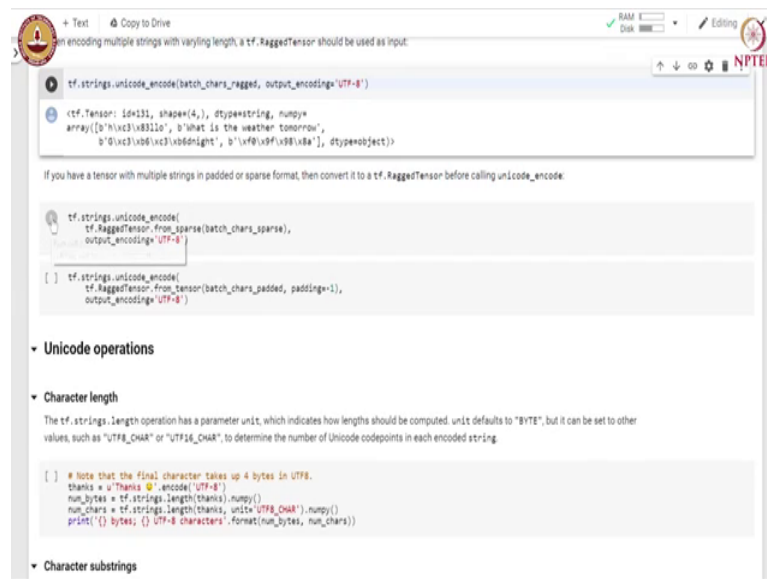
Below the code cell, there is a text box with the following text:

When encoding multiple strings with varying length, a `tf.RaggedTensor` should be used as input.

```
[ ] tf.strings.unicode_encode(batch_chars_ragged, output_encoding='UTF-8')
```

When encoding multiple strings with the same length a `tf.tensor` may be used as an input.
When encoding multiple strings with varying length a `tf.RaggedTensor` should be used as an input.

(Refer Slide Time: 25:10)



encoding multiple strings with varying length, a `tf.RaggedTensor` should be used as input:

```
tf.strings.unicode_encode(batch_chars_ragged, output_encoding='UTF-8')
```

`<tf.Tensor: id=131, shape=(4,), dtype=string, numpy=`
`array([b'\xc3\x83llo', b'what is the weather tomorrow',`
 `b'\xc3\x83\xbdontgnt', b'\xf0\x9f\x98\xba'], dtype=object)>`

If you have a tensor with multiple strings in padded or sparse format, then convert it to a `tf.RaggedTensor` before calling `unicode_encode`:

```
tf.strings.unicode_encode(
    tf.RaggedTensor.from_sparse(batch_chars_sparse),
    output_encoding='UTF-8')

[ ] tf.strings.unicode_encode(
    tf.RaggedTensor.from_tensor(batch_chars_padded, padding=1),
    output_encoding='UTF-8')
```

Unicode operations

Character length

The `tf.strings.length` operation has a parameter `unit`, which indicates how lengths should be computed. `unit` defaults to "BYTE", but it can be set to other values, such as "UTF8_CHAR" or "UTF16_CHAR", to determine the number of Unicode codepoints in each encoded string.

```
[ ] # Note that the final character takes up 4 bytes in UTF8.
    thanks = u'Thanks @!'.encode('UTF-8')
    num_bytes = tf.strings.length(thanks).numpy()
    num_chars = tf.strings.length(thanks, unit='UTF8_CHAR').numpy()
    print('{} bytes; {} UTF-8 characters'.format(num_bytes, num_chars))
```

Character substrings

If we have tensors with multiple strings in padded or sparse format then convert it to a `RaggedTensor` before calling unicode encode.

(Refer Slide Time: 25:37)



```
tf.strings.unicode_encode(
    tf.RaggedTensor.from_tensor(batch_chars_padded, padding=1),
    output_encoding='UTF-8')
```

`<tf.Tensor: id=289, shape=(4,), dtype=string, numpy=`
`array([b'\xc3\x83llo', b'what is the weather tomorrow',`
 `b'\xc3\x83\xbdontgnt', b'\xf0\x9f\x98\xba'], dtype=object)>`

Unicode operations

Character length

The `tf.strings.length` operation has a parameter `unit`, which indicates how lengths should be computed. `unit` defaults to "BYTE", but it can be set to other values, such as "UTF8_CHAR" or "UTF16_CHAR", to determine the number of Unicode codepoints in each encoded string.

```
[ ] # Note that the final character takes up 4 bytes in UTF8.
    thanks = u'Thanks @!'.encode('UTF-8')
    num_bytes = tf.strings.length(thanks).numpy()
    num_chars = tf.strings.length(thanks, unit='UTF8_CHAR').numpy()
    print('{} bytes; {} UTF-8 characters')
```

11 bytes; 8 UTF-8 characters

Character substrings

Similarly, the `tf.strings.substr` operation accepts the `unit` parameter, and uses it to determine what kind of offsets the `pos` and `len` parameters contain.

```
[ ] # default: unit='BYTE'. With len=1, we return a single byte.
    tf.strings.substr(thanks, pos=7, len=1).numpy()
```

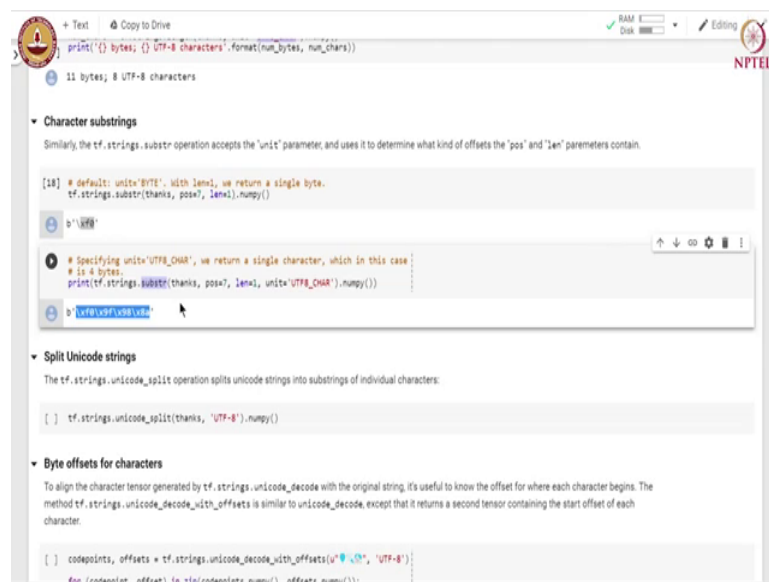
```
[ ] # Specifying unit='UTF8_CHAR', we return a single character, which in this case
    # is 4 bytes.
    print(tf.strings.substr(thanks, pos=7, len=1, unit='UTF8_CHAR').numpy())
```

Let us look at some of the unicode operations. Let us see how to find the length of the unicode string. So, the `tf.strings.length` has a parameter `unit` which indicates how length should be computed. `unit` defaults to bytes, but it can be set to other values such as

UTF8_CHAR or UTF16_CHAR to determine the number of unicode code points in each encoded strings.

So, let us take an example where we take thanks and smiley emoji and encode that using UTF-8 encoding store that in the thanks and we can use the length without any argument. So, we get number of bytes and if you want to find out number of characters we specify the unit as UTF8_CHAR and get the length. This particular string is encoded with 11 bytes and there are 8 UTF8_CHAR.

(Refer Slide Time: 27:06)



The screenshot shows a Jupyter Notebook interface with the following content:

- Code cell:

```
print('{} bytes; {} UTF-8 characters'.format(num_bytes, num_chars))
```


Output:

```
11 bytes; 8 UTF-8 characters
```
- Section: **Character substrings**
Text: Similarly, the `tf.strings.substr` operation accepts the `'unit'` parameter, and uses it to determine what kind of offsets the `'pos'` and `'len'` parameters contain.
Code:

```
[18] # default: unit='BYTE', with len=1, we return a single byte.  
tf.strings.substr(thanks, pos=7, len=1).numpy()
```


Output:

```
b'\U0001f60a'
```


Text:

```
# Specifying unit='UTF8_CHAR', we return a single character, which in this case  
# is 4 bytes.  
print(tf.strings.substr(thanks, pos=7, len=1, unit='UTF8_CHAR').numpy())
```


Output:

```
b'\U0001f60a\U0001f60a'
```
- Section: **Split Unicode strings**
Text: The `tf.strings.unicode_split` operation splits unicode strings into substrings of individual characters:
Code:

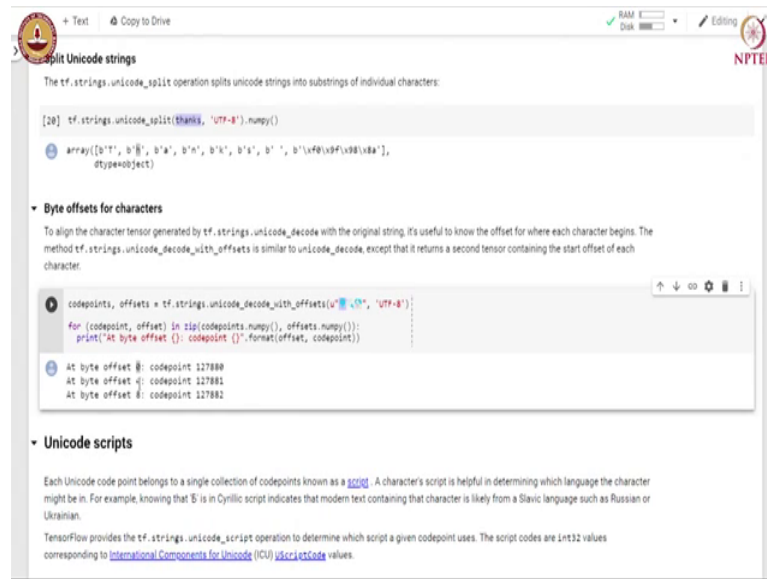
```
[ ] tf.strings.unicode_split(thanks, 'UTF-8').numpy()
```
- Section: **Byte offsets for characters**
Text: To align the character tensor generated by `tf.strings.unicode_decode` with the original string, it's useful to know the offset for where each character begins. The method `tf.strings.unicode_decode_with_offsets` is similar to `unicode_decode`, except that it returns a second tensor containing the start offset of each character.
Code:

```
[ ] codepoints, offsets = tf.strings.unicode_decode_with_offsets(u'\U0001f60a\U0001f60a', 'UTF-8')  
for (codepoint, offset) in zip(codepoints.numpy(), offsets.numpy()):
```

We can use `tf.strings.substr()` operation to get character sub strings. It also accepts the unit parameter and uses it to determine what kind of offsets the position and length parameter contain. Here the default unit is byte with length 1 we return a single byte. Here the unit is UTF8_CHAR and we return a single character which in this case is 4 bytes.

So, are you getting the difference between substrings here and substring here we are not specifying unit. So, by default byte is taken as the unit. Here we get a single byte and here we get single character which is 4 bytes. We can also split the unicode string using `unicode_split()` operation.

(Refer Slide Time: 28:31)



The screenshot shows a Jupyter Notebook interface with a title bar that includes a 'Copy to Drive' button and a status bar with 'RAM', 'Disk', and 'Editing' indicators. The notebook content is titled 'Split Unicode strings' and explains that the `tf.strings.unicode_split` operation splits unicode strings into substrings of individual characters. It shows a code cell with the following code:

```
[20] tf.strings.unicode_split('thanks', 'UTF-8').numpy()
```

The output is a NumPy array of byte strings: `array([b't', b'h', b'a', b'n', b'k', b's', b' ', b'\xf0\x9f\x98\x8a'], dtype=object)`. Below this, a section titled 'Byte offsets for characters' explains how to align the character tensor generated by `tf.strings.unicode_decode` with the original string. It introduces the `tf.strings.unicode_decode_with_offsets` method, which returns a second tensor containing the start offset of each character. A code cell demonstrates this with an emoji string:

```
codepoints, offsets = tf.strings.unicode_decode_with_offsets('👋👋👋', 'UTF-8')
for (codepoint, offset) in zip(codepoints.numpy(), offsets.numpy()):
    print('At byte offset {}: codepoint {}'.format(offset, codepoint))
```

The output shows the byte offsets and code points for the three emoji characters:

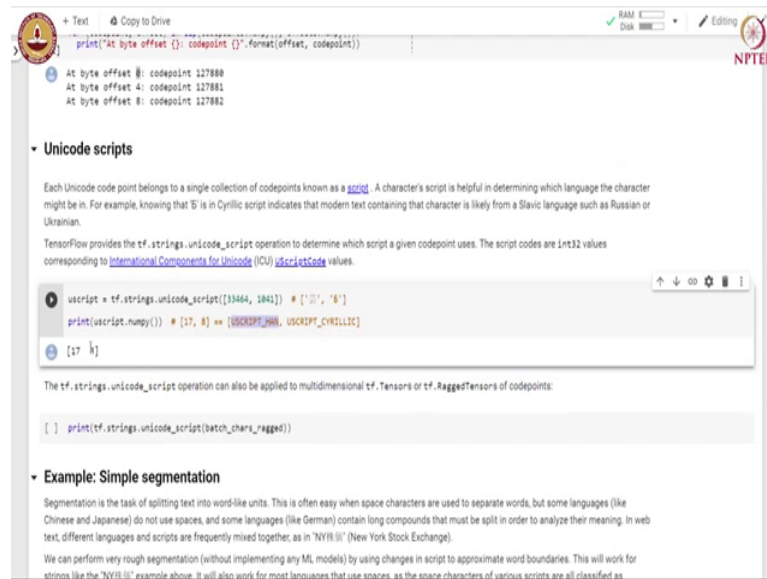
```
At byte offset 0: codepoint 127880
At byte offset 4: codepoint 127881
At byte offset 8: codepoint 127882
```

A final section titled 'Unicode scripts' explains that each Unicode code point belongs to a single collection of codepoints known as a 'script'. It mentions that a character's script is helpful in determining which language the character might be in, using the example of the Cyrillic script indicating modern text containing that character is likely from a Slavic language such as Russian or Ukrainian. It also notes that TensorFlow provides the `tf.strings.unicode_script` operation to determine which script a given codepoint uses, with script codes being `int32` values corresponding to [International Components for Unicode \(ICU\) UScriptCode](#) values.

To align the character tensor generated by `tf.strings.unicode_decode` with offset to align character tensors generated by `tf.strings.unicode_decode` with the original strings. It is useful to know the offsets for where each character begins for where each character begins the method `tf.strings.unicode_decode_with_offsets` is similar to `unicode_decode` except that it returns a second tensor containing the start offset of each character.

So, let us use `unicode_decode_with_offset` on a string of emoji which is encoded in UTF-8 it returns code points and the offsets. We print the offset and the code point. You can see that the first emoji is represented with code point 127880, the second emoji is encoded with this particular code point and third one is encoded with this particular point. We also get corresponding byte offsets of each of the emoji in the original string.

(Refer Slide Time: 30:37)



The screenshot shows a Jupyter Notebook interface with a code cell and its output. The code cell contains the following Python code:

```
print("At byte offset {}: codepoint {}".format(offset, codepoint))
```

The output shows three lines of text:

```
At byte offset 0: codepoint 127888
At byte offset 4: codepoint 127881
At byte offset 8: codepoint 127882
```

Below the code cell, there is a section titled "Unicode scripts" which explains that each Unicode code point belongs to a single collection of codepoints known as a script. It also mentions that TensorFlow provides the `tf.strings.unicode_script` operation to determine which script a given codepoint uses. The script codes are `int32` values corresponding to [International Components for Unicode \(ICU\) uScriptCode](#) values.

The code cell below this section contains the following Python code:

```
uscript = tf.strings.unicode_script([33464, 1041]) * ['?', 'B']
print(uscript.numpy())
```

The output shows the following array:

```
[17  8]
```

Below the output, there is a note that the `tf.strings.unicode_script` operation can also be applied to multidimensional `tf.Tensors` or `tf.RaggedTensors` of codepoints:

```
[ ] print(tf.strings.unicode_script(batch_chars_ragged))
```

Finally, there is a section titled "Example: Simple segmentation" which explains that segmentation is the task of splitting text into word-like units. It also mentions that some languages (like Chinese and Japanese) do not use spaces, and some languages (like German) contain long compounds that must be split in order to analyze their meaning. In web text, different languages and scripts are frequently mixed together, as in "NYSE III" (New York Stock Exchange).

We can perform very rough segmentation (without implementing any ML models) by using changes in script to approximate word boundaries. This will work for strings like the "NYSE III" example above. It will also work for most languages that use spaces, as the space characters of various scripts are all classified as

Each unicode code point belong to a single collection of code points known as a script. A character scripts is helpful in determine which language the character might be in for example, knowing that this particular character is a Cyrillic script indicates that modern text containing that character is likely from a Slavic language such as Russian or Ukrainian. TensorFlow provides `unicode_script` operation to determine which script a given code point uses. The script codes are so you can see that they have script 17 and 8 17 corresponds to Han Chinese and it corresponds to Cyrillic.