Introduction to Tensors and Variables

Learning Objectives

1. Understand Basic and Advanced Tensor Concepts
2. Understand Single-Axis and Multi-Axis Indexing
3. Create Tensors and Variables

Introduction

In this notebook, we look at tensors, which are multi-dimensional arrays with a uniform type (called a dtype). You can see all supported dtypes at tf.dtypes.DType. If you're familiar with NumPy, tensors are (kind of) like np.arrays. All tensors are immutable like python numbers and strings: you can never update the contents of a tensor, only create a new one.

We also look at variables, a `tf.Variable` represents a tensor whose value can be changed by running operations (ops) on it. Specific ops allow you to read and modify the values of this tensor. Higher level libraries like `tf.keras` use `tf.Variable` to store model parameters.

Each learning objective will correspond to a **#TODO** in the notebook where you will complete the notebook cell's code before running. Refer to the solution for reference.

Load necessary libraries

We will start by importing the necessary libraries for this lab.

In [1]:
```
import tensorflow as tf
import numpy as np

print("TensorFlow version: ",tf.version.VERSION)
```

```
TensorFlow version:  2.6.0
```

Lab Task 1: Understand Basic and Advanced Tensor Concepts

Basics

Let's create some basic tensors.

Here is a "scalar" or "rank-0" tensor . A scalar contains a single value, and no "axes".

In [2]:
```
# This will be an int32 tensor by default; see "dtypes" below.
rank_0_tensor = tf.constant(4)
print(rank_0_tensor)
```

```
tf.Tensor(4, shape=(), dtype=int32)
```

```
2021-09-19 12:25:19.162038: I tensorflow/core/common_runtime/process_util.c
c:146] Creating new thread pool with default inter op setting: 2. Tune usin
g inter_op_parallelism_threads for best performance.
```

A "vector" or "rank-1" tensor is like a list of values. A vector has 1-axis:

In [3]:
```
# Let's make this a float tensor.
rank_1_tensor = tf.constant([2.0, 3.0, 4.0])
print(rank_1_tensor)
```

```
tf.Tensor([2. 3. 4.], shape=(3,), dtype=float32)
```

A "matrix" or "rank-2" tensor has 2-axes:

In [4]:
```
# If we want to be specific, we can set the dtype (see below) at creation
rank_2_tensor = tf.constant([[1, 2],
                             [3, 4],
                             [5, 6]], dtype= # TODO 1a
                                      # TODO: Your code goes here.
print(rank_2_tensor)
```

```
  File "/tmp/ipykernel_3663/4102821594.py", line 6
    print(rank_2_tensor)
                       ^
SyntaxError: unexpected EOF while parsing
```

A scalar, shape: `[]` A vector, shape: `[3]` A matrix, shape: `[3, 2]`

Tensors may have more axes, here is a tensor with 3-axes:

In [6]:
```
# There can be an arbitrary number of
# axes (sometimes called "dimensions")
rank_3_tensor = tf.constant([
  [[0, 1, 2, 3, 4],
   [5, 6, 7, 8, 9]],
  [[10, 11, 12, 13, 14],
   [15, 16, 17, 18, 19]],
  [[20, 21, 22, 23, 24],
   [25, 26, 27, 28, 29]],])

print(rank_3_tensor)
```

```
tf.Tensor(
[[[ 0  1  2  3  4]
  [ 5  6  7  8  9]]

 [[10 11 12 13 14]
  [15 16 17 18 19]]

 [[20 21 22 23 24]
  [25 26 27 28 29]]], shape=(3, 2, 5), dtype=int32)
```

There are many ways you might visualize a tensor with more than 2-axes.

A 3-axis tensor, shape: `[3, 2, 5]`

You can convert a tensor to a NumPy array either using `np.array` or the

`tensor.numpy` method:

In [8]:
```
# Convert a tensor to a NumPy array using `np.array` method
# TODO 1b
# TODO -- Your code here.
np.array(rank_3_tensor)
```

Out[8]:
```
array([[[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9]],

       [[10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]],

       [[20, 21, 22, 23, 24],
        [25, 26, 27, 28, 29]]], dtype=int32)
```

In [10]:
```
# Convert a tensor to a NumPy array using `tensor.numpy` method
# TODO 1c
# TODO -- Your code here.
np.array(rank_3_tensor)
```

Out[10]:
```
array([[[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9]],

       [[10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]],

       [[20, 21, 22, 23, 24],
        [25, 26, 27, 28, 29]]], dtype=int32)
```

Tensors often contain floats and ints, but have many other types, including:

- complex numbers
- strings

The base `tf.Tensor` class requires tensors to be "rectangular"---that is, along each axis, every element is the same size. However, there are specialized types of Tensors that can handle different shapes:

- ragged (see RaggedTensor below)
- sparse (see SparseTensor below)

We can do basic math on tensors, including addition, element-wise multiplication, and matrix multiplication.

In [11]:
```
a = tf.constant([[1, 2],
                 [3, 4]])
b = tf.constant([[1, 1],
                 [1, 1]]) # Could have also said `tf.ones([2,2])`

print(tf.add(a, b), "\n")
print(tf.multiply(a, b), "\n")
print(tf.matmul(a, b), "\n")
```

```
tf.Tensor(
[[2 3]
 [4 5]], shape=(2, 2), dtype=int32)
```

```
tf.Tensor(
[[1 2]
 [3 4]], shape=(2, 2), dtype=int32)

tf.Tensor(
[[3 3]
 [7 7]], shape=(2, 2), dtype=int32)
```

In [12]:

```
print(a + b, "\n") # element-wise addition
print(a * b, "\n") # element-wise multiplication
print(a @ b, "\n") # matrix multiplication
```

```
tf.Tensor(
[[2 3]
 [4 5]], shape=(2, 2), dtype=int32)

tf.Tensor(
[[1 2]
 [3 4]], shape=(2, 2), dtype=int32)

tf.Tensor(
[[3 3]
 [7 7]], shape=(2, 2), dtype=int32)
```

Tensors are used in all kinds of operations (ops).

In [14]:

```
c = tf.constant([[4.0, 5.0], [10.0, 1.0]])

# Find the largest value
print(tf.reduce_max(c))
# TODO 1d
# Find the index of the largest value
print(tf.argmax(c))
# TODO -- Your code here.
# Compute the softmax
# TODO -- Your code here.
print(tf.nn.softmax(c))
```

```
tf.Tensor(10.0, shape=(), dtype=float32)
tf.Tensor([1 0], shape=(2,), dtype=int64)
tf.Tensor(
[[2.6894143e-01 7.3105860e-01]
 [9.9987662e-01 1.2339458e-04]], shape=(2, 2), dtype=float32)
```

About shapes

Tensors have shapes. Some vocabulary:

- **Shape**: The length (number of elements) of each of the dimensions of a tensor.
- **Rank**: Number of tensor dimensions. A scalar has rank 0, a vector has rank 1, a matrix is rank 2.
- **Axis** or **Dimension**: A particular dimension of a tensor.
- **Size**: The total number of items in the tensor, the product shape vector

Note: Although you may see reference to a "tensor of two dimensions", a rank-2 tensor does not usually describe a 2D space.

Tensors and `tf.TensorShape` objects have convenient properties for accessing these:

In [15]:
```
rank_4_tensor = tf.zeros([3, 2, 4, 5])
```

A rank-4 tensor, shape: `[3, 2, 4, 5]`

In [16]:
```
print("Type of every element:", rank_4_tensor.dtype)
print("Number of dimensions:", rank_4_tensor.ndim)
print("Shape of tensor:", rank_4_tensor.shape)
print("Elements along axis 0 of tensor:", rank_4_tensor.shape[0])
print("Elements along the last axis of tensor:", rank_4_tensor.shape[-1])
print("Total number of elements (3*2*4*5): ", tf.size(rank_4_tensor).numpy
```

```
Type of every element: <dtype: 'float32'>
Number of dimensions: 4
Shape of tensor: (3, 2, 4, 5)
Elements along axis 0 of tensor: 3
Elements along the last axis of tensor: 5
Total number of elements (3*2*4*5):  120
```

While axes are often referred to by their indices, you should always keep track of the meaning of each. Often axes are ordered from global to local: The batch axis first, followed by spatial dimensions, and features for each location last. This way feature vectors are contiguous regions of memory.

Typical axis order

Lab Task 2: Understand Single-Axis and Multi-Axis Indexing

Single-axis indexing

TensorFlow follow standard python indexing rules, similar to indexing a list or a string in python, and the bacic rules for numpy indexing.

- indexes start at `0`
- negative indices count backwards from the end
- colons, `:`, are used for slices `start:stop:step`

In [17]:
```
rank_1_tensor = tf.constant([0, 1, 1, 2, 3, 5, 8, 13, 21, 34])
print(rank_1_tensor.numpy())
```

```
[ 0  1  1  2  3  5  8 13 21 34]
```

Indexing with a scalar removes the dimension:

In [18]:
```
print("First:", rank_1_tensor[0].numpy())
print("Second:", rank_1_tensor[1].numpy())
print("Last:", rank_1_tensor[-1].numpy())
```

```
First: 0
Second: 1
Last: 34
```

Indexing with a `:` slice keeps the dimension:

In [19]:
```
print("Everything:", rank_1_tensor[:].numpy())
print("Before 4:", rank_1_tensor[:4].numpy())
print("From 4 to the end:", rank_1_tensor[4:].numpy())
print("From 2, before 7:", rank_1_tensor[2:7].numpy())
print("Every other item:", rank_1_tensor[::2].numpy())
print("Reversed:", rank_1_tensor[::-1].numpy())
```

```
Everything: [ 0  1  1  2  3  5  8 13 21 34]
Before 4: [0 1 1 2]
From 4 to the end: [ 3  5  8 13 21 34]
From 2, before 7: [1 2 3 5 8]
Every other item: [ 0  1  3  8 21]
Reversed: [34 21 13  8  5  3  2  1  1  0]
```

Multi-axis indexing

Higher rank tensors are indexed by passing multiple indices.

The single-axis exact same rules as in the single-axis case apply to each axis independently.

In [21]:
```
print(rank_3_tensor.numpy())
```

```
[[[ 0  1  2  3  4]
  [ 5  6  7  8  9]]

 [[10 11 12 13 14]
  [15 16 17 18 19]]

 [[20 21 22 23 24]
  [25 26 27 28 29]]]
```

Passing an integer for each index the result is a scalar.

In [22]:
```
# Pull out a single value from a 2-rank tensor
print(rank_3_tensor[1, 1].numpy())
```

```
[15 16 17 18 19]
```

You can index using any combination integers and slices:

In [25]:
```
# Get row and column tensors
print("Second row:", rank_3_tensor[1, :].numpy())
print("Second column:", rank_3_tensor[:, 1].numpy())
print("Last row:", rank_3_tensor[-1, :].numpy())
print("First item in last column:", rank_3_tensor[0, -1].numpy())
print("Skip the first row:")
print(rank_3_tensor[1:, :].numpy(), "\n")
```

```
Second row: [[10 11 12 13 14]
 [15 16 17 18 19]]
Second column: [[ 5  6  7  8  9]
 [15 16 17 18 19]
 [25 26 27 28 29]]
Last row: [[20 21 22 23 24]
```

```
 [25 26 27 28 29]]
First item in last column: [5 6 7 8 9]
Skip the first row:
[[[10 11 12 13 14]
  [15 16 17 18 19]]

 [[20 21 22 23 24]
  [25 26 27 28 29]]]
```

Here is an example with a 3-axis tensor:

In [26]:
```
print(rank_3_tensor[:, :, 4])
```

```
tf.Tensor(
[[ 4  9]
 [14 19]
 [24 29]], shape=(3, 2), dtype=int32)
```

Selecting the last feature across all locations in each example in the batch

Manipulating Shapes

Reshaping a tensor is of great utility.

The `tf.reshape` operation is fast and cheap as the underlying data does not need to be duplicated.

In [27]:
```
# Shape returns a `TensorShape` object that shows the size on each dimensi
var_x = tf.Variable(tf.constant([[1], [2], [3]]))
print(var_x.shape)
```

```
(3, 1)
```

In [28]:
```
# You can convert this object into a Python list, too
print(var_x.shape.as_list())
```

```
[3, 1]
```

You can reshape a tensor into a new shape. Reshaping is fast and cheap as the underlying data does not need to be duplicated.

In [31]:
```
# `tf.cast` casts a tensor to a new type.
# TODO 2b
the_f64_tensor = tf.constant([2.2, 3.3, 4.4], dtype=tf.float64)
the_f16_tensor = tf.cast(the_f64_tensor, dtype=tf.float16)
# Now, let's cast to an uint8 and lose the decimal precision
the_u8_tensor = tf.cast(the_f16_tensor, dtype=tf.uint8)
print(the_u8_tensor)
```

```
tf.Tensor([2 3 4], shape=(3,), dtype=uint8)
```

In [32]:
```
print(var_x.shape)
print(reshaped.shape)
```

```
(3, 1)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_3663/2699597244.py in <module>
      1 print(var_x.shape)
----> 2 print(reshaped.shape)
```

The data maintains it's layout in memory and a new tensor is created, with the requested shape, pointing to the same data. TensorFlow uses C-style "row-major" memory ordering, where incrementing the right-most index corresponds to a single step in memory.

In [33]:
```
print(rank_3_tensor)
```

```
tf.Tensor(
[[[ 0  1  2  3  4]
  [ 5  6  7  8  9]]

 [[10 11 12 13 14]
  [15 16 17 18 19]]

 [[20 21 22 23 24]
  [25 26 27 28 29]]], shape=(3, 2, 5), dtype=int32)
```

If you flatten a tensor you can see what order it is laid out in memory.

In [34]:
```
# A `-1` passed in the `shape` argument says "Whatever fits".
print(tf.reshape(rank_3_tensor, [-1]))
```

```
tf.Tensor(
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29], shape=(30,), dtype=int32)
```

Typically the only reasonable uses of `tf.reshape` are to combine or split adjacent axes (or add/remove `1`s).

For this 3x2x5 tensor, reshaping to (3x2)x5 or 3x(2x5) are both reasonable things to do, as the slices do not mix:

In [35]:
```
print(tf.reshape(rank_3_tensor, [3*2, 5]), "\n")
print(tf.reshape(rank_3_tensor, [3, -1]))
```

```
tf.Tensor(
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]
 [25 26 27 28 29]], shape=(6, 5), dtype=int32)

tf.Tensor(
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24 25 26 27 28 29]], shape=(3, 10), dtype=int32)
```

Some good reshapes.

Reshaping will "work" for any new shape with the same total number of elements, but it will not do anything useful if you do not respect the order of the axes.

Swapping axes in `tf.reshape` does not work, you need `tf.transpose` for that.

In [36]:
```
# Bad examples: don't do this

# You can't reorder axes with reshape.
print(tf.reshape(rank_3_tensor, [2, 3, 5]), "\n")

# This is a mess
print(tf.reshape(rank_3_tensor, [5, 6]), "\n")

# This doesn't work at all
try:
  tf.reshape(rank_3_tensor, [7, -1])
except Exception as e: print(e)
```

```
tf.Tensor(
[[[ 0  1  2  3  4]
  [ 5  6  7  8  9]
  [10 11 12 13 14]]

 [[15 16 17 18 19]
  [20 21 22 23 24]
  [25 26 27 28 29]]], shape=(2, 3, 5), dtype=int32)

tf.Tensor(
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]
 [24 25 26 27 28 29]], shape=(5, 6), dtype=int32)

Input to reshape is a tensor with 30 values, but the requested shape requir
es a multiple of 7 [Op:Reshape]
```

Some bad reshapes.

You may run across not-fully-specified shapes. Either the shape contains a `None` (a dimension's length is unknown) or the shape is `None` (the rank of the tensor is unknown).

Except for tf.RaggedTensor, this will only occur in the context of TensorFlow's, symbolic, graph-building APIs:

- tf.function
- The keras functional API.

More on `DTypes`

To inspect a `tf.Tensor` 's data type use the `Tensor.dtype` property.

When creating a `tf.Tensor` from a Python object you may optionally specify the datatype.

If you don't, TensorFlow chooses a datatype that can represent your data. TensorFlow converts Python integers to `tf.int32` and python floating point numbers to `tf.float32` . Otherwise TensorFlow uses the same rules NumPy uses when converting to arrays.

You can cast from type to type.

In [38]:
```
# `tf.cast` casts a tensor to a new type.
# TODO 2b
the_f64_tensor = tf.constant([2.2, 3.3, 4.4], dtype=tf.float64)
the_f16_tensor = tf.cast(the_f64_tensor, dtype=tf.float16)
# Now, let's cast to an uint8 and lose the decimal precision
the_u8_tensor = tf.cast(the_f16_tensor, dtype=tf.uint8)
print(the_u8_tensor)
```

```
tf.Tensor([2 3 4], shape=(3,), dtype=uint8)
```

Broadcasting

Broadcasting is a concept borrowed from the equivalent feature in NumPy. In short, under certain conditions, smaller tensors are "stretched" automatically to fit larger tensors when running combined operations on them.

The simplest and most common case is when you attempt to multiply or add a tensor to a scalar. In that case, the scalar is broadcast to be the same shape as the other argument.

In [39]:
```
x = tf.constant([1, 2, 3])

y = tf.constant(2)
z = tf.constant([2, 2, 2])
# All of these are the same computation
print(tf.multiply(x, 2))
print(x * y)
print(x * z)
```

```
tf.Tensor([2 4 6], shape=(3,), dtype=int32)
tf.Tensor([2 4 6], shape=(3,), dtype=int32)
tf.Tensor([2 4 6], shape=(3,), dtype=int32)
```

Likewise, 1-sized dimensions can be stretched out to match the other arguments. Both arguments can be stretched in the same computation.

In this case a 3x1 matrix is element-wise multiplied by a 1x4 matrix to produce a 3x4 matrix. Note how the leading 1 is optional: The shape of y is `[4]`.

In [40]:
```
# These are the same computations
x = tf.reshape(x,[3,1])
y = tf.range(1, 5)
print(x, "\n")
print(y, "\n")
print(tf.multiply(x, y))
```

```
tf.Tensor(
[[1]
 [2]
 [3]], shape=(3, 1), dtype=int32)

tf.Tensor([1 2 3 4], shape=(4,), dtype=int32)

tf.Tensor(
[[ 1  2  3  4]
```

```
 [ 2  4  6  8]
 [ 3  6  9 12]], shape=(3, 4), dtype=int32)
```

A broadcasted add: a `[3, 1]` times a `[1, 4]` gives a `[3,4]`

Here is the same operation without broadcasting:

In [41]:
```
x_stretch = tf.constant([[1, 1, 1, 1],
                         [2, 2, 2, 2],
                         [3, 3, 3, 3]])

y_stretch = tf.constant([[1, 2, 3, 4],
                         [1, 2, 3, 4],
                         [1, 2, 3, 4]])

print(x_stretch * y_stretch)  # Again, operator overloading
```

```
tf.Tensor(
[[ 1  2  3  4]
 [ 2  4  6  8]
 [ 3  6  9 12]], shape=(3, 4), dtype=int32)
```

Most of the time, broadcasting is both time and space efficient, as the broadcast operation never materializes the expanded tensors in memory.

You see what broadcasting looks like using `tf.broadcast_to`.

In [42]:
```
print(tf.broadcast_to(tf.constant([1, 2, 3]), [3, 3]))
```

```
tf.Tensor(
[[1 2 3]
 [1 2 3]
 [1 2 3]], shape=(3, 3), dtype=int32)
```

Unlike a mathematical op, for example, `broadcast_to` does nothing special to save memory. Here, you are materializing the tensor.

It can get even more complicated. This section of Jake VanderPlas's book *Python Data Science Handbook* shows more broadcasting tricks (again in NumPy).

tf.convert_to_tensor

Most ops, like `tf.matmul` and `tf.reshape` take arguments of class `tf.Tensor`. However, you'll notice in the above case, we frequently pass Python objects shaped like tensors.

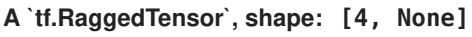
Most, but not all, ops call `convert_to_tensor` on non-tensor arguments. There is a registry of conversions, and most object classes like NumPy's `ndarray`, `TensorShape`, Python lists, and `tf.Variable` will all convert automatically.

See `tf.register_tensor_conversion_function` for more details, and if you have your own type you'd like to automatically convert to a tensor.

Ragged Tensors

A tensor with variable numbers of elements along some axis is called "ragged". Use `tf.ragged.RaggedTensor` for ragged data.

For example, This cannot be represented as a regular tensor:



In [43]:
```
ragged_list = [
    [0, 1, 2, 3],
    [4, 5],
    [6, 7, 8],
    [9]]
```

In [44]:
```
try:
  tensor = tf.constant(ragged_list)
except Exception as e: print(e)
```

```
Can't convert non-rectangular Python sequence to Tensor.
```

Instead create a `tf.RaggedTensor` using `tf.ragged.constant`:

In [46]:
```
# `tf.ragged.constant` constructs a constant RaggedTensor from a nested Py
# TODO 2c
ragged_tensor = tf.ragged.constant(ragged_list)
print(ragged_tensor)
```

```
<tf.RaggedTensor [[0, 1, 2, 3], [4, 5], [6, 7, 8], [9]]>
```

The shape of a `tf.RaggedTensor` contains unknown dimensions:

In [47]:
```
print(ragged_tensor.shape)
```

```
(4, None)
```

String tensors

`tf.string` is a `dtype`, which is to say we can represent data as strings (variable-length byte arrays) in tensors.

The strings are atomic and cannot be indexed the way Python strings are. The length of the string is not one of the dimensions of the tensor. See `tf.strings` for functions to manipulate them.

Here is a scalar string tensor:

In [48]:
```
# Tensors can be strings, too here is a scalar string.
scalar_string_tensor = tf.constant("Gray wolf")
print(scalar_string_tensor)
```

```
tf.Tensor(b'Gray wolf', shape=(), dtype=string)
```

And a vector of strings:

In [49]:
```
# If we have two string tensors of different lengths, this is OK.
tensor_of_strings = tf.constant(["Gray wolf",
                                 "Quick brown fox",
                                 "Lazy dog"])
# Note that the shape is (2,), indicating that it is 2 x unknown.
print(tensor_of_strings)
```

```
tf.Tensor([b'Gray wolf' b'Quick brown fox' b'Lazy dog'], shape=(3,), dtype=
string)
```

In the above printout the `b` prefix indicates that `tf.string` dtype is not a unicode string, but a byte-string. See the Unicode Tutorial for more about working with unicode text in TensorFlow.

If you pass unicode characters they are utf-8 encoded.

In [50]:
```
tf.constant("🥳👍")
```

Out[50]:
```
<tf.Tensor: shape=(), dtype=string, numpy=b'\xf0\x9f\xa5\xb3\xf0\x9f\x91\x8
d'>
```

Some basic functions with strings can be found in `tf.strings`, including `tf.strings.split`.

In [51]:
```
# We can use split to split a string into a set of tensors
print(tf.strings.split(scalar_string_tensor, sep=" "))
```

```
tf.Tensor([b'Gray' b'wolf'], shape=(2,), dtype=string)
```

In [52]:
```
# ...but it turns into a `RaggedTensor` if we split up a tensor of strings
# as each string might be split into a different number of parts.
print(tf.strings.split(tensor_of_strings))
```

```
<tf.RaggedTensor [[b'Gray', b'wolf'], [b'Quick', b'brown', b'fox'], [b'Lazy
', b'dog']]>
```

Three strings split, shape: [3, None]

And `tf.string.to_number`:

In [53]:
```
text = tf.constant("1 10 100")
print(tf.strings.to_number(tf.strings.split(text, " ")))
```

```
tf.Tensor([  1.  10. 100.], shape=(3,), dtype=float32)
```

Although you can't use `tf.cast` to turn a string tensor into numbers, you can convert it into bytes, and then into numbers.

In [54]:
```
byte_strings = tf.strings.bytes_split(tf.constant("Duck"))
byte_ints = tf.io.decode_raw(tf.constant("Duck"), tf.uint8)
print("Byte strings:", byte_strings)
print("Bytes:", byte_ints)
```

```
Byte strings: tf.Tensor([b'D' b'u' b'c' b'k'], shape=(4,), dtype=string)
Bytes: tf.Tensor([ 68 117  99 107], shape=(4,), dtype=uint8)
```

In [55]:
```
# Or split it up as unicode and then decode it
unicode_bytes = tf.constant("アヒル 🦆")
unicode_char_bytes = tf.strings.unicode_split(unicode_bytes, "UTF-8")
unicode_values = tf.strings.unicode_decode(unicode_bytes, "UTF-8")

print("\nUnicode bytes:", unicode_bytes)
print("\nUnicode chars:", unicode_char_bytes)
print("\nUnicode values:", unicode_values)
```

```
Unicode bytes: tf.Tensor(b'\xe3\x82\xa2\xe3\x83\x92\xe3\x83\xab \xf0\x9f\xa
6\x86', shape=(), dtype=string)

Unicode chars: tf.Tensor([b'\xe3\x82\xa2' b'\xe3\x83\x92' b'\xe3\x83\xab' b
' ' b'\xf0\x9f\xa6\x86'], shape=(5,), dtype=string)

Unicode values: tf.Tensor([ 12450  12498  12523     32 129414], shape=(5,),
dtype=int32)
```

The `tf.string` dtype is used for all raw bytes data in TensorFlow. The `tf.io` module contains functions for converting data to and from bytes, including decoding images and parsing csv.

Sparse tensors

Sometimes, your data is sparse, like a very wide embedding space. TensorFlow supports `tf.sparse.SparseTensor` and related operations to store sparse data efficiently.

In [57]:
```
# Sparse tensors store values by index in a memory-efficient manner
# TODO 2d
sparse_tensor = tf.sparse.SparseTensor(indices=[[0, 0], [1, 2]],
                                       values=[1, 2],
                                       dense_shape=[3, 4])
print(sparse_tensor, "\n")

# We can convert sparse tensors to dense
print(tf.sparse.to_dense(sparse_tensor))
```

```
SparseTensor(indices=tf.Tensor(
[[0 0]
 [1 2]], shape=(2, 2), dtype=int64), values=tf.Tensor([1 2], shape=(2,), dt
ype=int32), dense_shape=tf.Tensor([3 4], shape=(2,), dtype=int64))

tf.Tensor(
[[1 0 0 0]
 [0 0 2 0]
 [0 0 0 0]], shape=(3, 4), dtype=int32)
```

Lab Task 3: Introduction to Variables

A TensorFlow **variable** is the recommended way to represent shared, persistent state your program manipulates. This guide covers how to create, update, and manage instances of `tf.Variable` in TensorFlow.

Variables are created and tracked via the `tf.Variable` class. A `tf.Variable` represents a tensor whose value can be changed by running ops on it. Specific ops allow you to read and modify the values of this tensor. Higher level libraries like `tf.keras` use `tf.Variable` to store model parameters.

Setup

This notebook discusses variable placement. If you want to see on what device your variables are placed, uncomment this line.

In [58]:
```
import tensorflow as tf

# Uncomment to see where your variables get placed (see below)
# tf.debugging.set_log_device_placement(True)
```

Create a variable

To create a variable, provide an initial value. The `tf.Variable` will have the same `dtype` as the initialization value.

In [61]:
```
# TODO 3a
# Let's set an initial value
my_tensor = tf.constant([[1.0, 2.0], [3.0, 4.0]])
# `tf.Variable` will have the same as `dtype`
my_variable = tf.Variable(my_tensor)

# Variables can be all kinds of types, just like tensors
bool_variable = tf.Variable([False, False, False, True])
complex_variable = tf.Variable([5 + 4j, 6 + 1j])
```

A variable looks and acts like a tensor, and, in fact, is a data structure backed by a `tf.Tensor` . Like tensors, they have a `dtype` and a shape, and can be exported to NumPy.

In [62]:
```
print("Shape: ",my_variable.shape)
print("DType: ",my_variable.dtype)
print("As NumPy: ", my_variable.numpy)
```

```
Shape:  (2, 2)
DType:  <dtype: 'float32'>
As NumPy:  <bound method BaseResourceVariable.numpy of <tf.Variable 'Variab
le:0' shape=(2, 2) dtype=float32, numpy=
array([[1., 2.],
       [3., 4.]], dtype=float32)>>
```

Most tensor operations work on variables as expected, although variables cannot be reshaped.

In [63]:
```
print("A variable:",my_variable)
print("\nViewed as a tensor:", tf.convert_to_tensor(my_variable))
print("\nIndex of highest value:", tf.argmax(my_variable))

# This creates a new tensor; it does not reshape the variable.
print("\nCopying and reshaping: ", tf.reshape(my_variable, ([1,4])))
```

```
A variable: <tf.Variable 'Variable:0' shape=(2, 2) dtype=float32, numpy=
array([[1., 2.],
       [3., 4.]], dtype=float32)>

Viewed as a tensor: tf.Tensor(
[[1. 2.]
 [3. 4.]], shape=(2, 2), dtype=float32)

Index of highest value: tf.Tensor([1 1], shape=(2,), dtype=int64)

Copying and reshaping:  tf.Tensor([[1. 2. 3. 4.]], shape=(1, 4), dtype=floa
t32)
```

As noted above, variables are backed by tensors. You can reassign the tensor using `tf.Variable.assign`. Calling `assign` does not (usually) allocate a new tensor; instead, the existing tensor's memory is reused.

In [64]:
```
a = tf.Variable([2.0, 3.0])
# This will keep the same dtype, float32
a.assign([1, 2])
# Not allowed as it resizes the variable:
try:
  a.assign([1.0, 2.0, 3.0])
except Exception as e: print(e)
```

```
Cannot assign to variable Variable:0 due to variable shape (2,) and value s
hape (3,) are incompatible
```

If you use a variable like a tensor in operations, you will usually operate on the backing tensor.

Creating new variables from existing variables duplicates the backing tensors. Two variables will not share the same memory.

In [65]:
```
a = tf.Variable([2.0, 3.0])
# Create b based on the value of a
b = tf.Variable(a)
a.assign([5, 6])

# a and b are different
print(a.numpy())
print(b.numpy())

# There are other versions of assign
print(a.assign_add([2,3]).numpy())  # [7. 9.]
print(a.assign_sub([7,9]).numpy())  # [0. 0.]
```

```
[5. 6.]
[2. 3.]
```

```
[7. 9.]
[0. 0.]
```

Lifecycles, naming, and watching

In Python-based TensorFlow, `tf.Variable` instance have the same lifecycle as other Python objects. When there are no references to a variable it is automatically deallocated.

Variables can also be named which can help you track and debug them. You can give two variables the same name.

In [66]:
```
# Create a and b; they have the same value but are backed by different ten
a = tf.Variable(my_tensor, name="Mark")
# A new variable with the same name, but different value
# Note that the scalar add is broadcast
b = tf.Variable(my_tensor + 1, name="Mark")

# These are elementwise-unequal, despite having the same name
print(a == b)
```

```
tf.Tensor(
[[False False]
 [False False]], shape=(2, 2), dtype=bool)
```

Variable names are preserved when saving and loading models. By default, variables in models will acquire unique variable names automatically, so you don't need to assign them yourself unless you want to.

Although variables are important for differentiation, some variables will not need to be differentiated. You can turn off gradients for a variable by setting `trainable` to false at creation. An example of a variable that would not need gradients is a training step counter.

In [67]:
```
step_counter = tf.Variable(1, trainable=False)
```

Placing variables and tensors

For better performance, TensorFlow will attempt to place tensors and variables on the fastest device compatible with its `dtype` . This means most variables are placed on a GPU if one is available.

However, we can override this. In this snippet, we can place a float tensor and a variable on the CPU, even if a GPU is available. By turning on device placement logging (see Setup), we can see where the variable is placed.

Note: Although manual placement works, using distribution strategies can be a more convenient and scalable way to optimize your computation.

If you run this notebook on different backends with and without a GPU you will see different logging. *Note that logging device placement must be turned on at the start of the session.*

In [68]:
```
with tf.device('CPU:0'):

  # Create some tensors
  a = tf.Variable([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
  b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])
  c = tf.matmul(a, b)

print(c)
```

```
tf.Tensor(
[[22. 28.]
 [49. 64.]], shape=(2, 2), dtype=float32)
```

It's possible to set the location of a variable or tensor on one device and do the computation on another device. This will introduce delay, as data needs to be copied between the devices.

You might do this, however, if you had multiple GPU workers but only want one copy of the variables.

In [69]:
```
with tf.device('CPU:0'):
  a = tf.Variable([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
  b = tf.Variable([[1.0, 2.0, 3.0]])

with tf.device('GPU:0'):
  # Element-wise multiply
  k = a * b

print(k)
```

```
tf.Tensor(
[[ 1.  4.  9.]
 [ 4. 10. 18.]], shape=(2, 3), dtype=float32)
```

Note: Because `tf.config.set_soft_device_placement` is turned on by default, even if you run this code on a device without a GPU, it will still run and the multiplication step happen on the CPU.

For more on distributed training, see our guide.

Next steps

To understand how variables are typically used, see our guide on automatic distribution.