



Getting started with TensorFlow tutorial: A guide to the fundamentals

What is TensorFlow?

[TensorFlow](#) is an open-source end-to-end machine learning library for preprocessing data, modelling data and serving models (getting them into the hands of others).

Why use TensorFlow?

Rather than building machine learning and deep learning models from scratch, it's more likely you'll use a library such as TensorFlow. This is because it contains many of the most common machine learning functions you'll want to use.

What we're going to cover

TensorFlow is vast. But the main premise is simple: turn data into numbers (tensors) and build machine learning algorithms to find patterns in them.

In this notebook we cover some of the most fundamental TensorFlow operations, more specifically:

- Introduction to tensors (creating tensors)
- Getting information from tensors (tensor attributes)
- Manipulating tensors (tensor operations)
- Tensors and NumPy
- Using @tf.function (a way to speed up your regular Python functions)
- Using GPUs with TensorFlow
- Exercises to try

Things to note:

- Many of the conventions here will happen automatically behind the scenes (when you build a model) but it's worth knowing so if you see any of these things, you know what's happening.
- For any TensorFlow function you see, it's important to be able to check it out in the documentation, for example, going to the Python API docs for all functions and searching for what you need: https://www.tensorflow.org/api_docs/python/ (don't worry if this seems overwhelming at first, with enough practice, you'll get used to navigating the documentaiton).

Introduction to Tensors

If you've ever used NumPy, [tensors](#) are kind of like NumPy arrays (we'll see more on this later).

For the sake of this notebook and going forward, you can think of a tensor as a multi-dimensional numerical representation (also referred to as n-dimensional, where n can be any number) of something. Where something can be almost anything you can imagine:

- It could be numbers themselves (using tensors to represent the price of houses).
- It could be an image (using tensors to represent the pixels of an image).
- It could be text (using tensors to represent words).
- Or it could be some other form of information (or data) you want to represent with numbers.

The main difference between tensors and NumPy arrays (also an n-dimensional array of numbers) is that tensors can be used on [GPUs \(graphical processing units\)](#) and [TPUs \(tensor processing units\)](#).

The benefit of being able to run on GPUs and TPUs is faster computation, this means, if we wanted to find patterns in the numerical representations of our data, we can generally find them faster using GPUs and TPUs.

Okay, we've been talking enough about tensors, let's see them.

The first thing we'll do is import TensorFlow under the common alias `tf`.

In [1]:

```
# Import TensorFlow
import tensorflow as tf
print(tf.__version__) # find the version number (should be 2.x+)
```

2.7.0

Creating Tensors with `tf.constant()`

As mentioned before, in general, you usually won't create tensors yourself. This is because TensorFlow has modules built-in (such as `tf.io` and `tf.data`) which are able to read your data sources and automatically convert them to tensors and then later on, neural network models will process these for us.

But for now, because we're getting familiar with tensors themselves and how to manipulate them, we'll see how we can create them ourselves.

We'll begin by using `tf.constant()`.

In [2]:

```
# Create a scalar (rank 0 tensor)
scalar = tf.constant(7)
scalar
```

Out[2]:

`<tf.Tensor: shape=(), dtype=int32, numpy=7>`

A scalar is known as a rank 0 tensor. Because it has no dimensions (it's just a number).

 **Note:** For now, you don't need to know too much about the different ranks of tensors (but we will see more on this later). The important point is knowing tensors

can have an unlimited range of dimensions (the exact amount will depend on what data you're representing).

In [3]:

```
# Check the number of dimensions of a tensor (ndim stands for number of dimensions)
scalar.ndim
```

Out[3]: 0

In [4]:

```
# Create a vector (more than 0 dimensions)
vector = tf.constant([10, 10])
vector
```

Out[4]: <tf.Tensor: shape=(2,), dtype=int32, numpy=array([10, 10])>

In [5]:

```
# Check the number of dimensions of our vector tensor
vector.ndim
```

Out[5]: 1

In [6]:

```
# Create a matrix (more than 1 dimension)
matrix = tf.constant([[10, 7],
                     [7, 10]])
matrix
```

Out[6]: <tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[10, 7],
 [7, 10]])>

In [7]:

```
matrix.ndim
```

Out[7]: 2

By default, TensorFlow creates tensors with either an `int32` or `float32` datatype.

This is known as [32-bit precision](#)) (the higher the number, the more precise the number, the more space it takes up on your computer).

In [8]:

```
# Create another matrix and define the datatype
another_matrix = tf.constant([[10., 7.],
                             [3., 2.],
                             [8., 9.]], dtype=tf.float16) # specify the datatype with
another_matrix
```

Out[8]: <tf.Tensor: shape=(3, 2), dtype=float16, numpy=
array([[10., 7.],
 [3., 2.],
 [8., 9.]], dtype=float16)>

In [9]:

```
# Even though another_matrix contains more numbers, its dimensions stay the same
another_matrix.ndim
```

Out[9]: 2

In [10]:

```
# How about a tensor? (more than 2 dimensions, although, all of the above items are also tensors)
tensor = tf.constant([[[1, 2, 3],
                      [4, 5, 6]],
                     [[7, 8, 9],
                      [10, 11, 12]],
                     [[13, 14, 15],
                      [16, 17, 18]]])
tensor
```

Out[10]: <tf.Tensor: shape=(3, 2, 3), dtype=int32, numpy=

```
array([[[ 1,  2,  3],
       [ 4,  5,  6]],

      [[ 7,  8,  9],
       [10, 11, 12]],

      [[13, 14, 15],
       [16, 17, 18]]])>
```

In [11]:

tensor.ndim

Out[11]: 3

This is known as a rank 3 tensor (3-dimensions), however a tensor can have an arbitrary (unlimited) amount of dimensions.

For example, you might turn a series of images into tensors with shape (224, 224, 3, 32), where:

- 224, 224 (the first 2 dimensions) are the height and width of the images in pixels.
- 3 is the number of colour channels of the image (red, green blue).
- 32 is the batch size (the number of images a neural network sees at any one time).

All of the above variables we've created are actually tensors. But you may also hear them referred to as their different names (the ones we gave them):

- **scalar**: a single number.
- **vector**: a number with direction (e.g. wind speed with direction).
- **matrix**: a 2-dimensional array of numbers.
- **tensor**: an n-dimensional array of numbers (where n can be any number, a 0-dimension tensor is a scalar, a 1-dimension tensor is a vector).

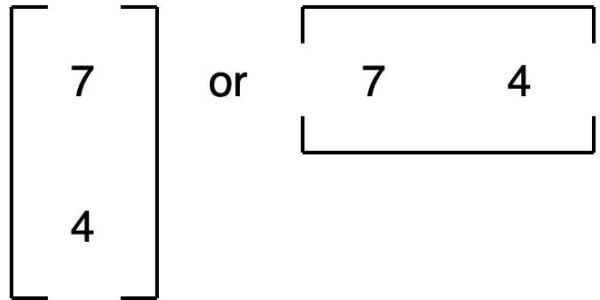
To add to the confusion, the terms matrix and tensor are often used interchangably.

Going forward since we're using TensorFlow, everything we refer to and use will be tensors.

For more on the mathematical difference between scalars, vectors and matrices see the [visual algebra post by Math is Fun](#).

Scalar

7

Vector**Matrix**

$$\begin{bmatrix} 7 & 10 \\ 4 & 3 \\ 5 & 1 \end{bmatrix}$$

Tensor

$$\begin{bmatrix} [7] & [4] & [0] & [1] \\ [1] & [9] & [2] & [3] \\ [5] & [6] & [8] & [8] \end{bmatrix}$$

Creating Tensors with `tf.Variable()`

You can also (although you likely rarely will, because often, when working with data, tensors are created for you automatically) create tensors using `tf.Variable()`.

The difference between `tf.Variable()` and `tf.constant()` is tensors created with `tf.constant()` are immutable (can't be changed, can only be used to create a new tensor), whereas, tensors created with `tf.Variable()` are mutable (can be changed).

```
In [12]: # Create the same tensor with tf.Variable() and tf.constant()
changeable_tensor = tf.Variable([10, 7])
unchangeable_tensor = tf.constant([10, 7])
changeable_tensor, unchangeable_tensor
```

```
Out[12]: (<tf.Variable 'Variable:0' shape=(2,) dtype=int32, numpy=array([10, 7])>,
<tf.Tensor: shape=(2,), dtype=int32, numpy=array([10, 7])>)
```

Now let's try to change one of the elements of the changable tensor.

```
In [13]: # Will error (requires the .assign() method)
changeable_tensor[0] = 7
changeable_tensor
```

```

TypeError                                         Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_16936/3008076862.py in <module>
      1 # Will error (requires the .assign() method)
----> 2 changeable_tensor[0] = 7
      3 changeable_tensor

```

TypeError: 'ResourceVariable' object does not support item assignment

To change an element of a `tf.Variable()` tensor requires the `assign()` method.

In [14]:

```

# Won't error
changeable_tensor[0].assign(7)
changeable_tensor

```

Out[14]:

```
<tf.Variable 'Variable:0' shape=(2,) dtype=int32, numpy=array([7, 7])>
```

Now let's try to change a value in a `tf.constant()` tensor.

In [15]:

```

# Will error (can't change tf.constant())
unchangeable_tensor[0].assign(7)
unchangeable_tensor

```

```

AttributeError                                         Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_16936/2797127693.py in <module>
      1 # Will error (can't change tf.constant())
----> 2 unchangeable_tensor[0].assign(7)
      3 unchangeable_tensor

```

```

~\anaconda3\lib\site-packages\tensorflow\python\framework\ops.py in __getattr__(self, name)
    440         from tensorflow.python.ops.numpy_ops import np_config
    441         np_config.enable_numpy_behavior()"".format(type(self).__name__, name))
--> 442     self.__getattribute__(name)
    443
    444     @staticmethod

```

AttributeError: 'tensorflow.python.framework.ops.EagerTensor' object has no attribute 'assign'

Which one should you use? `tf.constant()` or `tf.Variable()`?

It will depend on what your problem requires. However, most of the time, TensorFlow will automatically choose for you (when loading data or modelling data).

Creating random tensors

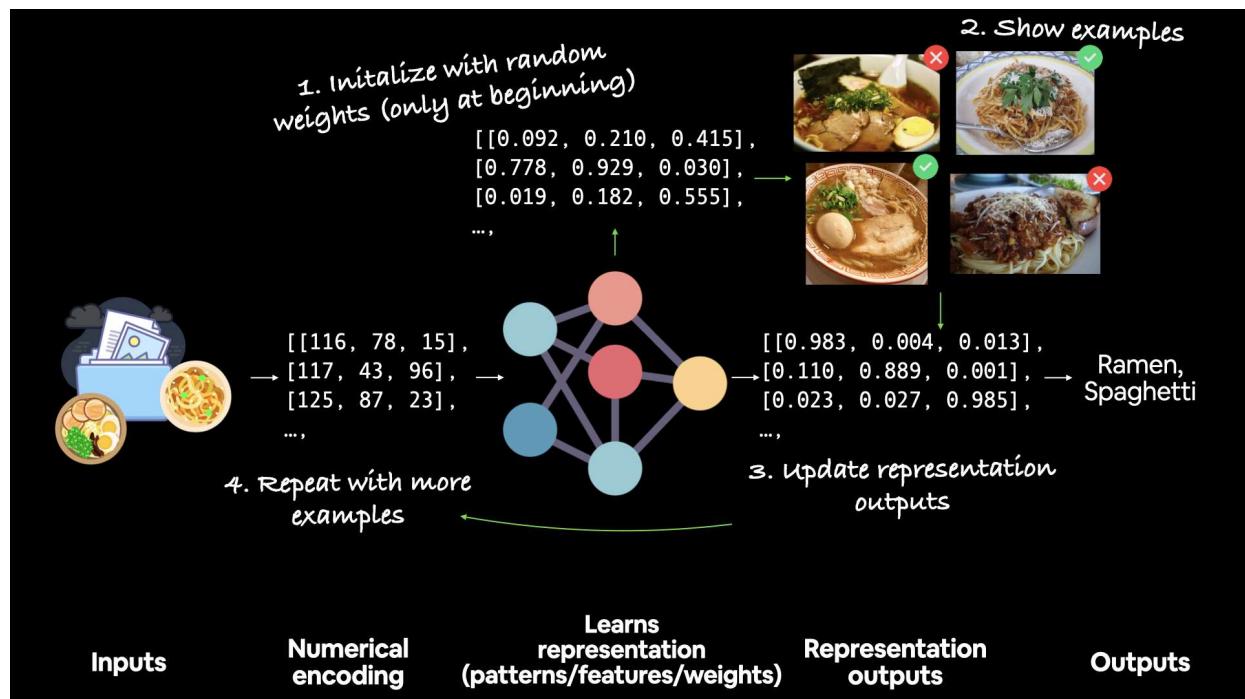
Random tensors are tensors of some arbitrary size which contain random numbers.

Why would you want to create random tensors?

This is what neural networks use to initialize their weights (patterns) that they're trying to learn in the data.

For example, the process of a neural network learning often involves taking a random n-dimensional array of numbers and refining them until they represent some kind of pattern (a compressed way to represent the original data).

How a network learns



A network learns by starting with random patterns (1) then going through demonstrative examples of data (2) whilst trying to update its random patterns to represent the examples (3).

We can create random tensors by using the `tf.random.Generator` class.

In [16]:

```
# Create two random (but the same) tensors
random_1 = tf.random.Generator.from_seed(42) # set the seed for reproducibility
random_1 = random_1.normal(shape=(3, 2)) # create tensor from a normal distribution
random_2 = tf.random.Generator.from_seed(42)
random_2 = random_2.normal(shape=(3, 2))

# Are they equal? yes
random_1, random_2, random_1 == random_2
```

Out[16]:

```
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[-0.7565803 , -0.06854702],
       [ 0.07595026, -1.2573844 ],
       [-0.23193763, -1.8107855 ]], dtype=float32)>,
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[-0.7565803 , -0.06854702],
       [ 0.07595026, -1.2573844 ],
       [-0.23193763, -1.8107855 ]], dtype=float32)>,
<tf.Tensor: shape=(3, 2), dtype=bool, numpy=
array([[ True,  True],
       [ True,  True],
       [ True,  True]])>
```

The random tensors we've made are actually [pseudorandom numbers](#) (they appear as random, but really aren't).

If we set a seed we'll get the same random numbers (if you've ever used NumPy, this is similar to `np.random.seed(42)`).

Setting the seed says, "hey, create some random numbers, but flavour them with X" (X is the seed).

What do you think will happen when we change the seed?

In [17]:

```
# Create two random (and different) tensors
random_3 = tf.random.Generator.from_seed(42)
random_3 = random_3.normal(shape=(3, 2))
random_4 = tf.random.Generator.from_seed(11)
random_4 = random_4.normal(shape=(3, 2))

# Check the tensors and see if they are equal
random_3, random_4, random_1 == random_3, random_3 == random_4
```

Out[17]:

```
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[-0.7565803 , -0.06854702],
       [ 0.07595026, -1.2573844 ],
       [-0.23193763, -1.8107855 ]], dtype=float32)>,
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[ 0.27305737, -0.29925638],
       [-0.3652325 ,  0.61883307],
       [-1.0130816 ,  0.28291714]], dtype=float32)>,
<tf.Tensor: shape=(3, 2), dtype=bool, numpy=
array([[ True,  True],
       [ True,  True],
       [ True,  True]])>,
<tf.Tensor: shape=(3, 2), dtype=bool, numpy=
array([[False, False],
       [False, False],
       [False, False]])>)
```

What if you wanted to shuffle the order of a tensor?

Wait, why would you want to do that?

Let's say you're working with 15,000 images of cats and dogs and the first 10,000 images were of cats and the next 5,000 were of dogs. This order could affect how a neural network learns (it may overfit by learning the order of the data), instead, it might be a good idea to move your data around.

In [18]:

```
# Shuffle a tensor (valuable for when you want to shuffle your data)
# Shuffle along the first dimension
not_shuffled = tf.constant([[10, 7],
                           [3, 4],
                           [2, 5]])
# Gets different results each time
tf.random.shuffle(not_shuffled)
```

Out[18]:

```
<tf.Tensor: shape=(3, 2), dtype=int32, numpy=
array([[ 2,  5],
       [ 3,  4],
       [10,  7]])>
```

In [19]:

```
# Shuffle in the same order every time using the seed parameter (won't actually be the
tf.random.shuffle(not_shuffled, seed=42)
```

```
Out[19]: <tf.Tensor: shape=(3, 2), dtype=int32, numpy=
array([[ 2,  5],
       [ 3,  4],
       [10,  7]])>
```

Wait... why didn't the numbers come out the same?

It's due to rule #4 of the `tf.random.set_seed()` documentation.

"4. If both the global and the operation seed are set: Both seeds are used in conjunction to determine the random sequence."

`tf.random.set_seed(42)` sets the global seed, and the `seed` parameter in `tf.random.shuffle(seed=42)` sets the operation seed.

Because, "Operations that rely on a random seed actually derive it from two seeds: the global and operation-level seeds. This sets the global seed."

```
In [20]: # Shuffle in the same order every time (Need to use global Level and operation Level se
# Set the global random seed
tf.random.set_seed(42)

# Set the operation random seed
tf.random.shuffle(not_shuffled, seed=42)
```

```
Out[20]: <tf.Tensor: shape=(3, 2), dtype=int32, numpy=
array([[10,  7],
       [ 3,  4],
       [ 2,  5]])>
```

```
In [21]: # Set the global random seed
tf.random.set_seed(42) # if you comment this out you'll get different results

# Set the operation random seed
tf.random.shuffle(not_shuffled)
```

```
Out[21]: <tf.Tensor: shape=(3, 2), dtype=int32, numpy=
array([[ 3,  4],
       [ 2,  5],
       [10,  7]])>
```

Other ways to make tensors

Though you might rarely use these (remember, many tensor operations are done behind the scenes for you), you can use `tf.ones()` to create a tensor of all ones and `tf.zeros()` to create a tensor of all zeros.

```
In [22]: # Make a tensor of all ones
tf.ones(shape=(3, 2))
```

```
In [22]: <tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[1., 1.],
       [1., 1.],
       [1., 1.]], dtype=float32)>
```

```
In [23]: # Make a tensor of all zeros
tf.zeros(shape=(3, 2))
```

```
Out[23]: <tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[0., 0.],
       [0., 0.],
       [0., 0.]], dtype=float32)>
```

You can also turn NumPy arrays into tensors.

Remember, the main difference between tensors and NumPy arrays is that tensors can be run on GPUs.

 **Note:** A matrix or tensor is typically represented by a capital letter (e.g. X or A) where as a vector is typically represented by a lowercase letter (e.g. y or b).

```
In [24]: import numpy as np
numpy_A = np.arange(1, 25, dtype=np.int32) # create a NumPy array between 1 and 25
A = tf.constant(numpy_A) # Create tensor using numpy array without specifying shape

# Create tensor using numpy array with specifying shape
B = tf.constant(numpy_A,
                shape=[2, 4, 3]) # note: the shape total (2*4*3) has to match the number
A, B
```

```
Out[24]: (<tf.Tensor: shape=(24,), dtype=int32, numpy=
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20, 21, 22, 23, 24]),>
<tf.Tensor: shape=(2, 4, 3), dtype=int32, numpy=
array([[[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]],
      [[13, 14, 15],
       [16, 17, 18],
       [19, 20, 21],
       [22, 23, 24]]])>)
```

Getting information from tensors (shape, rank, size)

There will be times when you'll want to get different pieces of information from your tensors, in particular, you should know the following tensor vocabulary:

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

You'll use these especially when you're trying to line up the shapes of your data to the shapes of your model. For example, making sure the shape of your image tensors are the same shape as your models input layer.

We've already seen one of these before using the `ndim` attribute. Let's see the rest.

In [25]:

```
# Create a rank 4 tensor (4 dimensions)
rank_4_tensor = tf.zeros([2, 3, 4, 5])
rank_4_tensor
```

Out[25]:

```
<tf.Tensor: shape=(2, 3, 4, 5), dtype=float32, numpy=
array([[[[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., 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., 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., 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.]]],
```

In [26]:

```
rank_4_tensor.shape, rank_4_tensor.ndim, tf.size(rank_4_tensor)
```

Out[26]:

```
(TensorShape([2, 3, 4, 5]), 4, <tf.Tensor: shape=(), dtype=int32, numpy=120>)
```

In [27]:

```
# Get various attributes of tensor
print("Datatype of every element:", rank_4_tensor.dtype)
print("Number of dimensions (rank):", 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 last axis of tensor:", rank_4_tensor.shape[-1])
print("Total number of elements (2*3*4*5):", tf.size(rank_4_tensor))
print("Total number of elements (2*3*4*5):", tf.size(rank_4_tensor).numpy() # .numpy()
```

Datatype of every element: <dtype: 'float32'>

```
Number of dimensions (rank): 4
Shape of tensor: (2, 3, 4, 5)
Elements along axis 0 of tensor: 2
Elements along last axis of tensor: 5
Total number of elements (2*3*4*5): tf.Tensor(120, shape=(), dtype=int32)
Total number of elements (2*3*4*5): 120
```

Indexing Tensors

You can also index tensors just like Python lists.

In [28]: rank_4_tensor

```
<tf.Tensor: shape=(2, 3, 4, 5), dtype=float32, numpy=
array([[[[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., 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., 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., 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., 0.],
          [0., 0., 0., 0., 0.],
          [0., 0., 0., 0., 0.],
          [0., 0., 0., 0., 0.]]], dtype=float32)>
```

In [29]: # Get the first 2 items of each dimension
rank_4_tensor[:2, :2, :2, :2]

```
<tf.Tensor: shape=(2, 2, 2, 2), dtype=float32, numpy=
array([[[[0., 0.],
          [0., 0.]],

         [[0., 0.],
          [0., 0.]]],
```

$$\begin{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \end{bmatrix}$$

```
[[0., 0.],
 [0., 0.]]], dtype=float32)>
```

In [30]:

```
# Get the first element from each dimension except for the final one
rank_4_tensor[:1, :1, :1, :]
```

Out[30]:

```
<tf.Tensor: shape=(1, 1, 1, 5), dtype=float32, numpy=array([[[[0., 0., 0., 0., 0.]]]], d
type=float32)>
```

In [31]:

```
# Get the first element from each dimension except for the 2nd one
rank_4_tensor[:1, :, :1, :1]
```

Out[31]:

```
<tf.Tensor: shape=(1, 3, 1, 1), dtype=float32, numpy=
array([[[[0.]],
 [[0.]]], dtype=float32)>
```

In [32]:

```
# Create a rank 2 tensor (2 dimensions)
rank_2_tensor = tf.constant([[10, 7],
 [3, 4]])
```

```
# Get the last item of each row
rank_2_tensor[:, -1]
```

Out[32]:

```
<tf.Tensor: shape=(2,), dtype=int32, numpy=array([7, 4])>
```

You can also add dimensions to your tensor whilst keeping the same information present using `tf.newaxis`.

In [33]:

```
# Add an extra dimension (to the end)
# Convert rows into separate cols
rank_3_tensor = rank_2_tensor[..., tf.newaxis] # in Python "..." means "all dimensions"
rank_2_tensor, rank_3_tensor # shape (2, 2), shape (2, 2, 1)
```

Out[33]:

```
(<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[10, 7],
 [3, 4]]),
<tf.Tensor: shape=(2, 2, 1), dtype=int32, numpy=
array([[[10]],
 [7]],
 [[ 3],
 [ 4]]])>)
```

You can achieve the same using `tf.expand_dims()`.

In [34]:

```
tf.expand_dims(rank_2_tensor, axis=-1) # "-1" means last axis
```

Out[34]:

```
<tf.Tensor: shape=(2, 2, 1), dtype=int32, numpy=
array([[[10]],
 [7]],
```

```
[[ 3],
 [ 4]])>
```

In [35]: `tf.expand_dims(rank_2_tensor, axis=0) # "0" means first axis`

Out[35]: `<tf.Tensor: shape=(1, 2, 2), dtype=int32, numpy=
array([[[10, 7],
 [3, 4]]])>`

Manipulating tensors (tensor operations)

Finding patterns in tensors (numerical representation of data) requires manipulating them.

Again, when building models in TensorFlow, much of this pattern discovery is done for you.

Basic operations

You can perform many of the basic mathematical operations directly on tensors using Python operators such as, `+`, `-`, `*`.

In [36]: `# You can add values to a tensor using the addition operator
tensor = tf.constant([[10, 7], [3, 4]])
tensor + 10`

Out[36]: `<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[20, 17],
 [13, 14]])>`

Since we used `tf.constant()`, the original tensor is unchanged (the addition gets done on a copy).

In [37]: `# Original tensor unchanged
tensor`

Out[37]: `<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[10, 7],
 [3, 4]])>`

Other operators also work.

In [38]: `# Multiplication (known as element-wise multiplication)
tensor * 10`

Out[38]: `<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[100, 70],
 [30, 40]])>`

In [39]: `# Subtraction
tensor - 10`

Out[39]: `<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[0, -3],
 [-7, -6]])>`

You can also use the equivalent TensorFlow function. Using the TensorFlow function (where possible) has the advantage of being sped up later down the line when running as part of a [TensorFlow graph](#).

```
In [40]: # Use the tensorflow function equivalent of the '*' (multiply) operator
tf.multiply(tensor, 10)
```

```
Out[40]: <tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[100, 70],
       [30, 40]])>
```

```
In [41]: # The original tensor is still unchanged
tensor
```

```
Out[41]: <tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[10, 7],
       [3, 4]])>
```

Matrix multiplication

One of the most common operations in machine learning algorithms is [matrix multiplication](#).

TensorFlow implements this matrix multiplication functionality in the `tf.matmul()` method.

The main two rules for matrix multiplication to remember are:

1. The inner dimensions must match:

- $(3, 5) @ (3, 5)$ won't work
- $(5, 3) @ (3, 5)$ will work
- $(3, 5) @ (5, 3)$ will work

2. The resulting matrix has the shape of the outer dimensions:

- $(5, 3) @ (3, 5) \rightarrow (5, 5)$
- $(3, 5) @ (5, 3) \rightarrow (3, 3)$

 **Note:** '@' in Python is the symbol for matrix multiplication.

```
In [42]: # Matrix multiplication in TensorFlow
print(tensor)
tf.matmul(tensor, tensor)
```

```
Out[42]: tf.Tensor(
[[10  7]
 [ 3  4]], shape=(2, 2), dtype=int32)
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[121, 98],
       [42, 37]])>
```

```
In [43]: # Matrix multiplication with Python operator '@'
tensor @ tensor
```

```
Out[43]: <tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[121, 98],
```

```
[ 42, 37]]))>
```

Both of these examples work because our `tensor` variable is of shape (2, 2).

What if we created some tensors which had mismatched shapes?

In [44]:

```
# Create (3, 2) tensor
X = tf.constant([[1, 2],
                 [3, 4],
                 [5, 6]])

# Create another (3, 2) tensor
Y = tf.constant([[7, 8],
                 [9, 10],
                 [11, 12]])

X, Y
```

Out[44]:

```
(<tf.Tensor: shape=(3, 2), dtype=int32, numpy=
 array([[1, 2],
        [3, 4],
        [5, 6]])>,
<tf.Tensor: shape=(3, 2), dtype=int32, numpy=
 array([[ 7,  8],
        [ 9, 10],
        [11, 12]])>)
```

In [45]:

```
# Try to matrix multiply them (will error)
X @ Y
```

```
-----  

InvalidArgumentError                                     Traceback (most recent call last)  

~\AppData\Local\Temp\ipykernel_16936/1079411915.py in <module>  

    1 # Try to matrix multiply them (will error)  

----> 2 X @ Y  
  

~\anaconda3\lib\site-packages\tensorflow\python\util\traceback_utils.py in error_handler  

(*args, **kwargs)  

    151     except Exception as e:  

    152         filtered_tb = _process_traceback_frames(e.__traceback__)  

--> 153         raise e.with_traceback(filtered_tb) from None  

    154     finally:  

    155         del filtered_tb  
  

~\anaconda3\lib\site-packages\tensorflow\python\framework\ops.py in raise_from_not_ok_st  

atus(e, name)  

    7105 def raise_from_not_ok_status(e, name):  

    7106     e.message += (" name: " + name if name is not None else "")  

-> 7107     raise core._status_to_exception(e) from None # pylint: disable=protected-acc  

ess  

    7108  

    7109
```

InvalidArgumentError: Matrix size-incompatible: In[0]: [3,2], In[1]: [3,2] [Op:MatMul]

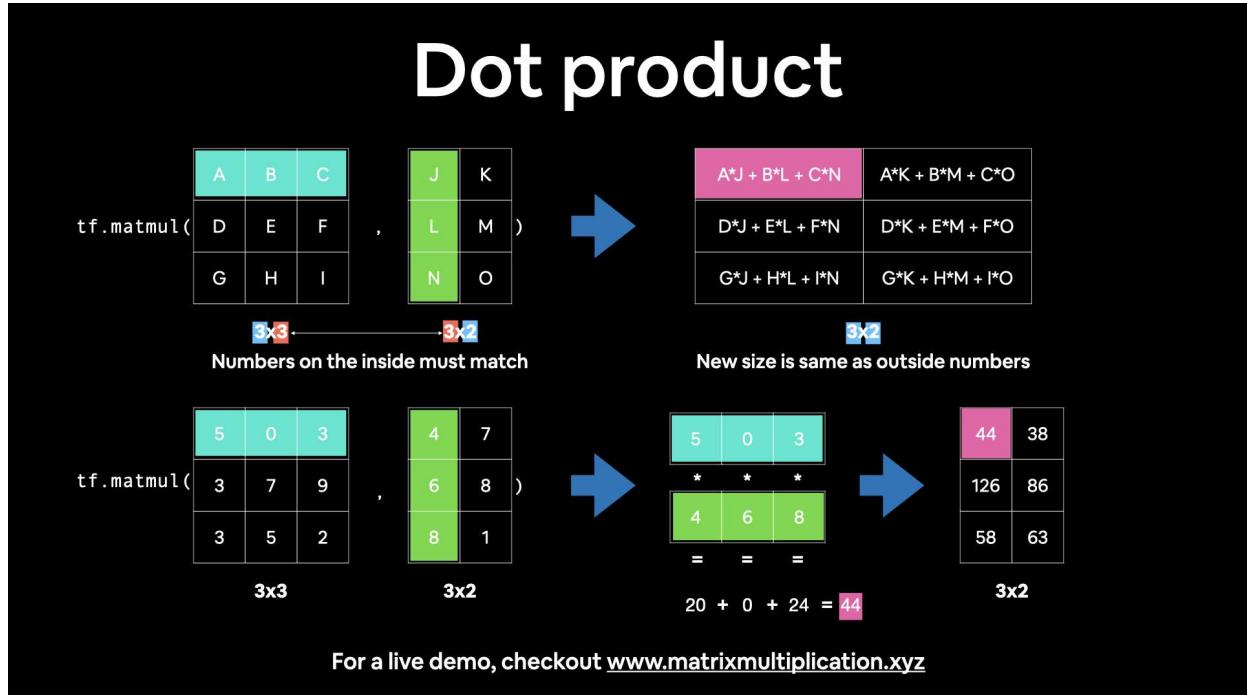
Trying to matrix multiply two tensors with the shape (3, 2) errors because the inner dimensions don't match.

We need to either:

- Reshape X to (2, 3) so it's (2, 3) @ (3, 2) .
- Reshape Y to (3, 2) so it's (3, 2) @ (2, 3) .

We can do this with either:

- `tf.reshape()` - allows us to reshape a tensor into a defined shape.
- `tf.transpose()` - switches the dimensions of a given tensor.



Let's try `tf.reshape()` first.

```
In [46]: # Example of reshape (3, 2) -> (2, 3)
tf.reshape(Y, shape=(2, 3))
```

```
Out[46]: <tf.Tensor: shape=(2, 3), dtype=int32, numpy=
array([[ 7,  8,  9],
       [10, 11, 12]])>
```

```
In [47]: # Try matrix multiplication with reshaped Y
X @ tf.reshape(Y, shape=(2, 3))
```

```
Out[47]: <tf.Tensor: shape=(3, 3), dtype=int32, numpy=
array([[ 27,  30,  33],
       [ 61,  68,  75],
       [ 95, 106, 117]])>
```

It worked, let's try the same with a reshaped `X`, except this time we'll use `tf.transpose()` and `tf.matmul()`.

```
In [48]: # Example of transpose (3, 2) -> (2, 3)
tf.transpose(X)
```

```
Out[48]: <tf.Tensor: shape=(2, 3), dtype=int32, numpy=
array([[1, 3, 5],
```

```
[2, 4, 6]])>
```

In [49]:

```
# Try matrix multiplication
tf.matmul(tf.transpose(X), Y)
```

Out[49]:

```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[ 89,  98],
       [116, 128]])>
```

In [50]:

```
# You can achieve the same result with parameters
tf.matmul(a=X, b=Y, transpose_a=True, transpose_b=False)
```

Out[50]:

```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[ 89,  98],
       [116, 128]])>
```

Notice the difference in the resulting shapes when tranposing X or reshaping Y .

This is because of the 2nd rule mentioned above:

- (3, 2) @ (2, 3) -> (2, 2) done with `tf.matmul(tf.transpose(X), Y)`
- (2, 3) @ (3, 2) -> (3, 3) done with `X @ tf.reshape(Y, shape=(2, 3))`

This kind of data manipulation is a reminder: you'll spend a lot of your time in machine learning and working with neural networks reshaping data (in the form of tensors) to prepare it to be used with various operations (such as feeding it to a model).

The dot product

Multiplying matrices by eachother is also referred to as the dot product.

You can perform the `tf.matmul()` operation using `tf.tensordot()` .

In [51]:

```
# Perform the dot product on X and Y (requires X to be transposed)
# Need to use axes as 1
tf.tensordot(tf.transpose(X), Y, axes=1)
```

Out[51]:

```
<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[ 89,  98],
       [116, 128]])>
```

You might notice that although using both `reshape` and `tranpose` work, you get different results when using each.

Let's see an example, first with `tf.transpose()` then with `tf.reshape()` .

In [52]:

```
# Perform matrix multiplication between X and Y (transposed)
tf.matmul(X, tf.transpose(Y))
```

Out[52]:

```
<tf.Tensor: shape=(3, 3), dtype=int32, numpy=
array([[ 23,  29,  35],
       [ 53,  67,  81],
       [ 83, 105, 127]])>
```

```
In [53]: # Perform matrix multiplication between X and Y (reshaped)
tf.matmul(X, tf.reshape(Y, (2, 3)))
```

```
Out[53]: <tf.Tensor: shape=(3, 3), dtype=int32, numpy=
array([[ 27,  30,  33],
       [ 61,  68,  75],
       [ 95, 106, 117]])>
```

Hmm... they result in different values.

Which is strange because when dealing with Y (a (3×2) matrix), reshaping to $(2, 3)$ and transposing it result in the same shape.

```
In [54]: # Check shapes of Y, reshaped Y and tranposed Y
Y.shape, tf.reshape(Y, (2, 3)).shape, tf.transpose(Y).shape
```

```
Out[54]: (TensorShape([3, 2]), TensorShape([2, 3]), TensorShape([2, 3]))
```

But calling `tf.reshape()` and `tf.transpose()` on Y don't necessarily result in the same values.

```
In [55]: # Check values of Y, reshape Y and tranposed Y
print("Normal Y:")
print(Y, "\n") # "\n" for newLine

print("Y reshaped to (2, 3):")
print(tf.reshape(Y, (2, 3)), "\n")

print("Y transposed:")
print(tf.transpose(Y))
```

```
Normal Y:
tf.Tensor(
[[ 7  8]
 [ 9 10]
 [11 12]], shape=(3, 2), dtype=int32)
```

```
Y reshaped to (2, 3):
tf.Tensor(
[[ 7  8  9]
 [10 11 12]], shape=(2, 3), dtype=int32)
```

```
Y transposed:
tf.Tensor(
[[ 7  9 11]
 [ 8 10 12]], shape=(2, 3), dtype=int32)
```

As you can see, the outputs of `tf.reshape()` and `tf.transpose()` when called on Y , even though they have the same shape, are different.

This can be explained by the default behaviour of each method:

- `tf.reshape()` - change the shape of the given tensor (first) and then insert values in order they appear (in our case, 7, 8, 9, 10, 11, 12).

- `tf.transpose()` - swap the order of the axes, by default the last axis becomes the first, however the order can be changed using the `perm` parameter.

So which should you use?

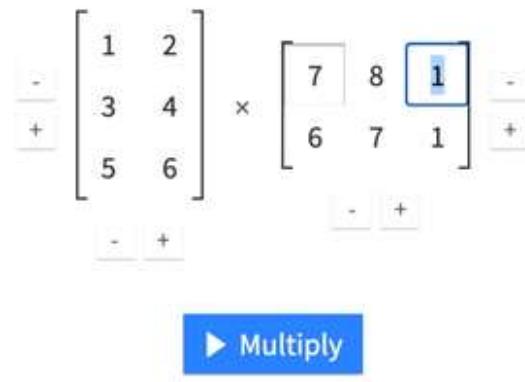
Again, most of the time these operations (when they need to be run, such as during the training a neural network, will be implemented for you).

But generally, whenever performing a matrix multiplication and the shapes of two matrices don't line up, you will transpose (not reshape) one of them in order to line them up.

Matrix multiplication tidbits

- If we transposed `Y`, it would be represented as \mathbf{Y}^T (note the capital T for transpose).
- Get an illustrative view of matrix multiplication [by Math is Fun](#).
- Try a hands-on demo of matrix multiplication: <http://matrixmultiplication.xyz/> (shown below).

Matrix Multiplication



Changing the datatype of a tensor

Sometimes you'll want to alter the default datatype of your tensor.

This is common when you want to compute using less precision (e.g. 16-bit floating point numbers vs. 32-bit floating point numbers).

Computing with less precision is useful on devices with less computing capacity such as mobile devices (because the less bits, the less space the computations require).

You can change the datatype of a tensor using `tf.cast()`.

In [56]:

```
# Create a new tensor with default datatype (float32)
B = tf.constant([1.7, 7.4])
```

```
# Create a new tensor with default datatype (int32)
C = tf.constant([1, 7])
B, C
```

```
In [56]: (<tf.Tensor: shape=(2,), dtype=float32, numpy=array([1.7, 7.4], dtype=float32)>,
<tf.Tensor: shape=(2,), dtype=int32, numpy=array([1, 7])>)
```

```
# Change from float32 to float16 (reduced precision)
B = tf.cast(B, dtype=tf.float16)
B
```

```
Out[57]: <tf.Tensor: shape=(2,), dtype=float16, numpy=array([1.7, 7.4], dtype=float16)>
```

```
# Change from int32 to float32
C = tf.cast(C, dtype=tf.float32)
C
```

```
Out[58]: <tf.Tensor: shape=(2,), dtype=float32, numpy=array([1., 7.], dtype=float32)>
```

Getting the absolute value

Sometimes you'll want the absolute values (all values are positive) of elements in your tensors.

To do so, you can use `tf.abs()`.

```
# Create tensor with negative values
D = tf.constant([-7, -10])
D
```

```
Out[59]: <tf.Tensor: shape=(2,), dtype=int32, numpy=array([-7, -10])>
```

```
# Get the absolute values
tf.abs(D)
```

```
Out[60]: <tf.Tensor: shape=(2,), dtype=int32, numpy=array([ 7, 10])>
```

Finding the min, max, mean, sum (aggregation)

You can quickly aggregate (perform a calculation on a whole tensor) tensors to find things like the minimum value, maximum value, mean and sum of all the elements.

To do so, aggregation methods typically have the syntax `reduce()_[action]`, such as:

- `tf.reduce_min()` - find the minimum value in a tensor.
- `tf.reduce_max()` - find the maximum value in a tensor (helpful for when you want to find the highest prediction probability).
- `tf.reduce_mean()` - find the mean of all elements in a tensor.
- `tf.reduce_sum()` - find the sum of all elements in a tensor.

- **Note:** typically, each of these is under the `math` module, e.g. `tf.math.reduce_min()` but you can use the alias `tf.reduce_min()`.

Let's see them in action.

In [61]:

```
# Create a tensor with 50 random values between 0 and 100
E = tf.constant(np.random.randint(low=0, high=100, size=50))
E
```

Out[61]:

```
<tf.Tensor: shape=(50,), dtype=int32, numpy=
array([79, 42, 89, 42, 55, 55, 71, 69, 83, 41, 37, 20, 78, 69, 69, 48, 11,
       82, 76, 68, 15, 72, 42, 16, 86, 33, 56, 85, 68, 28, 65, 67, 34, 78,
      16, 75, 57, 75, 68, 18, 51, 61,  8, 77, 49, 97, 38, 46, 46, 99])>
```

In [62]:

```
# Find the minimum
tf.reduce_min(E)
```

Out[62]:

```
<tf.Tensor: shape=(), dtype=int32, numpy=8>
```

In [63]:

```
# Find the minimum value
tf.reduce_min(E).numpy()
```

Out[63]:

```
8
```

In [64]:

```
# Find the maximum
tf.reduce_max(E)
```

Out[64]:

```
<tf.Tensor: shape=(), dtype=int32, numpy=99>
```

In [65]:

```
# Find the mean
tf.reduce_mean(E)
```

Out[65]:

```
<tf.Tensor: shape=(), dtype=int32, numpy=56>
```

In [66]:

```
# Find the sum
tf.reduce_sum(E)
```

Out[66]:

```
<tf.Tensor: shape=(), dtype=int32, numpy=2810>
```

Find Variance and Standard Deviation

In [67]:

```
# To find the variance import tensorflow_probability Library
import tensorflow_probability as tfp
tfp.stats.variance(E)
```

Out[67]:

```
<tf.Tensor: shape=(), dtype=int32, numpy=561>
```

In [68]:

```
# Another way of Finding the variance
# Should cast to float
```

```
tf.math.reduce_variance(tf.cast(E, dtype=tf.float32))
```

Out[68]: <tf.Tensor: shape=(), dtype=float32, numpy=561.12006>

In [69]: # Find the standard deviation

```
# Should cast to float
tf.math.reduce_std(tf.cast(E, dtype=tf.float32))
```

Out[69]: <tf.Tensor: shape=(), dtype=float32, numpy=23.687973>

Finding the positional maximum and minimum

How about finding the position a tensor where the maximum value occurs?

This is helpful when you want to line up your labels (say ['Green', 'Blue', 'Red']) with your prediction probabilities tensor (e.g. [0.98, 0.01, 0.01]).

In this case, the predicted label (the one with the highest prediction probability) would be 'Green' .

You can do the same for the minimum (if required) with the following:

- `tf.argmax()` - find the position of the maximum element in a given tensor.
- `tf.argmin()` - find the position of the minimum element in a given tensor.

In [70]:

```
# Create a tensor with 50 values between 0 and 1
F = tf.constant(np.random.random(50))
F
```

Out[70]:

```
<tf.Tensor: shape=(50,), dtype=float64, numpy=
array([0.01507729, 0.68378144, 0.13726867, 0.23399918, 0.69749883,
       0.11867872, 0.74713062, 0.42543588, 0.64184449, 0.88783455,
       0.00931243, 0.95476321, 0.18995871, 0.1743075 , 0.64614584,
       0.24450747, 0.49891097, 0.57303102, 0.20653672, 0.5692502 ,
       0.06353439, 0.80566423, 0.83971917, 0.04647004, 0.71471538,
       0.45603461, 0.85326355, 0.06171025, 0.9760033 , 0.45801514,
       0.67505343, 0.31051134, 0.49476384, 0.02779815, 0.46715377,
       0.27445725, 0.21785467, 0.36339867, 0.55091452, 0.7393417 ,
       0.83683926, 0.49906863, 0.50094414, 0.73872405, 0.00457606,
       0.21072207, 0.04181134, 0.78030725, 0.45431328, 0.90284913])>
```

In [71]:

```
# Find the index of maximum element in F
tf.argmax(F)
```

Out[71]:

```
<tf.Tensor: shape=(), dtype=int64, numpy=28>
```

In [72]:

```
# Find the index of minimum element in F
tf.argmin(F)
```

Out[72]:

```
<tf.Tensor: shape=(), dtype=int64, numpy=44>
```

```
In [73]: # Find the maximum element position of F
print(f"The index of maximum value of F: {tf.argmax(F).numpy()}")
print(f"The maximum value of F: {tf.reduce_max(F).numpy()}")
print(f"Using tf.argmax() to index F, the maximum value of F is: {F[tf.argmax(F)].numpy()}")
print(f"Are the two max values the same (they should be)? {F[tf.argmax(F)].numpy() == t}
```

The index of maximum value of F: 28
The maximum value of F: 0.9760033028708581
Using tf.argmax() to index F, the maximum value of F is: 0.9760033028708581
Are the two max values the same (they should be)? True

Squeezing a tensor (removing all single dimensions)

If you need to remove single-dimensions from a tensor (dimensions with size 1), you can use `tf.squeeze()`.

- `tf.squeeze()` - remove all dimensions of 1 from a tensor.

```
In [74]: # Create a rank 5 (5 dimensions) tensor of 50 numbers between 0 and 100
G = tf.constant(np.random.randint(0, 100, 50), shape=(1, 1, 1, 1, 50))
G.shape, G.ndim
```

Out[74]: (TensorShape([1, 1, 1, 1, 50]), 5)

```
In [75]: # Squeeze tensor G (remove all 1 dimensions)
G_squeezed = tf.squeeze(G)
G_squeezed.shape, G_squeezed.ndim
```

Out[75]: (TensorShape([50]), 1)

One-hot encoding

If you have a tensor of indices and would like to one-hot encode it, you can use `tf.one_hot()`.

You should also specify the `depth` parameter (the level which you want to one-hot encode to).

```
In [76]: # Create a list of indices
some_list = [0, 1, 2, 3]

# One hot encode them
tf.one_hot(some_list, depth=4)
```

```
Out[76]: <tf.Tensor: shape=(4, 4), dtype=float32, numpy=
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]], dtype=float32)>
```

You can also specify values for `on_value` and `off_value` instead of the default 0 and 1.

```
In [77]: # Specify custom values for on and off encoding
tf.one_hot(some_list, depth=4, on_value="We're live!", off_value="Offline")
```

```
Out[77]: <tf.Tensor: shape=(4, 4), dtype=string, numpy=
array([[b"We're live!", b'Offline', b'Offline', b'Offline'],
       [b'Offline', b"We're live!", b'Offline', b'Offline'],
       [b'Offline', b'Offline', b"We're live!", b'Offline'],
       [b'Offline', b'Offline', b'Offline', b"We're live!"]], dtype=object)>
```

Squaring, log, square root

Many other common mathematical operations you'd like to perform at some stage, probably exist.

Let's take a look at:

- `tf.square()` - get the square of every value in a tensor.
- `tf.sqrt()` - get the squareroot of every value in a tensor (**note**: the elements need to be floats or this will error).
- `tf.math.log()` - get the natural log of every value in a tensor (elements need to floats).

In [78]:

```
# Create a new tensor
H = tf.constant(np.arange(1, 10))
H
```

Out[78]:

```
<tf.Tensor: shape=(9,), dtype=int32, numpy=array([1, 2, 3, 4, 5, 6, 7, 8, 9])>
```

In [79]:

```
# Square it
tf.square(H)
```

Out[79]:

```
<tf.Tensor: shape=(9,), dtype=int32, numpy=array([ 1,  4,  9, 16, 25, 36, 49, 64, 81])>
```

In [80]:

```
# Find the squareroot (will error), needs to be non-integer
tf.sqrt(H)
```

```
InvalidArgumentError                                     Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_16936\2861151801.py in <module>
      1 # Find the squareroot (will error), needs to be non-integer
----> 2 tf.sqrt(H)

~\anaconda3\lib\site-packages\tensorflow\python\util\traceback_utils.py in error_handler
(*args, **kwargs)
    151     except Exception as e:
    152         filtered_tb = _process_traceback_frames(e.__traceback__)
--> 153         raise e.with_traceback(filtered_tb) from None
    154     finally:
    155         del filtered_tb

~\anaconda3\lib\site-packages\tensorflow\python\framework\ops.py in raise_from_not_ok_status(e, name)
    7105 def raise_from_not_ok_status(e, name):
    7106     e.message += (" name: " + name if name is not None else "")
-> 7107     raise core._status_to_exception(e) from None # pylint: disable=protected-access
    7108
    7109
```

```
InvalidArgumentError: Value for attr 'T' of int32 is not in the list of allowed values:
bfloating16, half, float, double, complex64, complex128
; NodeDef: {{node Sqrt}}; Op<name=Sqrt; signature=x:T -> y:T; attr=T:type,allowe
d=[DT_BFLOAT16, DT_HALF, DT_FLOAT, DT_DOUBLE, DT_COMPLEX64, DT_COMPLEX128]> [Op:Sqrt]
```

In [81]:

```
# Change H to float32
H = tf.cast(H, dtype=tf.float32)
H
```

Out[81]:

```
<tf.Tensor: shape=(9,), dtype=float32, numpy=array([1., 2., 3., 4., 5., 6., 7., 8., 9.], dtype=float32)>
```

In [82]:

```
# Find the square root
tf.sqrt(H)
```

Out[82]:

```
<tf.Tensor: shape=(9,), dtype=float32, numpy=
array([0.99999994, 1.4142134 , 1.7320508 , 1.9999999 , 2.236068 ,
       2.4494896 , 2.6457512 , 2.8284268 , 3.           ], dtype=float32)>
```

In [83]:

```
# Find the log (input also needs to be float)
tf.math.log(H)
```

Out[83]:

```
<tf.Tensor: shape=(9,), dtype=float32, numpy=
array([0.          , 0.6931472, 1.0986123, 1.3862944, 1.609438 , 1.7917595,
       1.9459102, 2.0794415, 2.1972246], dtype=float32)>
```

Manipulating tf.Variable tensors

Tensors created with `tf.Variable()` can be changed in place using methods such as:

- `.assign()` - assign a different value to a particular index of a variable tensor.
- `.add_assign()` - add to an existing value and reassign it at a particular index of a variable tensor.

In [84]:

```
# Create a variable tensor
I = tf.Variable(np.arange(0, 5))
I
```

Out[84]:

```
<tf.Variable 'Variable:0' shape=(5,) dtype=int32, numpy=array([0, 1, 2, 3, 4])>
```

In [85]:

```
# Assign the final value a new value of 50
I.assign([0, 1, 2, 3, 50])
```

Out[85]:

```
<tf.Variable 'UnreadVariable' shape=(5,) dtype=int32, numpy=array([ 0,  1,  2,  3, 50])>
```

In [86]:

```
# The change happens in place (the last value is now 50, not 4)
I
```

Out[86]:

```
<tf.Variable 'Variable:0' shape=(5,) dtype=int32, numpy=array([ 0,  1,  2,  3, 50])>
```

In [87]:

```
# Add 10 to every element in I
```

```
I.assign_add([10, 10, 10, 10, 10])
```

Out[87]: <tf.Variable 'UnreadVariable' shape=(5,) dtype=int32, numpy=array([10, 11, 12, 13, 60])>

```
# Again, the change happens in place
I
```

Out[88]: <tf.Variable 'Variable:0' shape=(5,) dtype=int32, numpy=array([10, 11, 12, 13, 60])>

Tensors and NumPy

We've seen some examples of tensors interact with NumPy arrays, such as, using NumPy arrays to create tensors.

Tensors can also be converted to NumPy arrays using:

- `np.array()` - pass a tensor to convert to an ndarray (NumPy's main datatype).
- `tensor.numpy()` - call on a tensor to convert to an ndarray.

Doing this is helpful as it makes tensors iterable as well as allows us to use any of NumPy's methods on them.

```
In [89]: # Create a tensor from a NumPy array
J = tf.constant(np.array([3., 7., 10.]))
J
```

Out[89]: <tf.Tensor: shape=(3,), dtype=float64, numpy=array([3., 7., 10.])>

```
In [90]: # Convert tensor J to NumPy with np.array()
np.array(J), type(np.array(J))
```

Out[90]: (array([3., 7., 10.]), numpy.ndarray)

```
In [91]: # Another way of converting tensor J
J.numpy(), type(J.numpy())
```

Out[91]: (array([3., 7., 10.]), numpy.ndarray)

By default tensors have `dtype=float32`, whereas NumPy arrays have `dtype=float64`.

This is because neural networks (which are usually built with TensorFlow) can generally work very well with less precision (32-bit rather than 64-bit).

```
In [92]: # Create a tensor from NumPy and from an array
numpy_J = tf.constant(np.array([3., 7., 10.])) # will be float64 (due to NumPy)
tensor_J = tf.constant([3., 7., 10.]) # will be float32 (due to being TensorFlow default)
numpy_J.dtype, tensor_J.dtype
```

(tf.float64, tf.float32)

Out[92]:

Using @tf.function

In your TensorFlow adventures, you might come across Python functions which have the decorator `@tf.function`.

If you aren't sure what Python decorators do, [read RealPython's guide on them](#).

But in short, decorators modify a function in one way or another.

In the `@tf.function` decorator case, it turns a Python function into a callable TensorFlow graph. Which is a fancy way of saying, if you've written your own Python function, and you decorate it with `@tf.function`, when you export your code (to potentially run on another device), TensorFlow will attempt to convert it into a fast(er) version of itself (by making it part of a computation graph).

For more on this, read the [Better performance with tf.function](#) guide.

In [93]:

```
# Create a simple function
def function(x, y):
    return x ** 2 + y

x = tf.constant(np.arange(0, 10))
y = tf.constant(np.arange(10, 20))
function(x, y)
```

Out[93]:

```
<tf.Tensor: shape=(10,), dtype=int32, numpy=array([ 10,  12,  16,  22,  30,  40,  52,  6
6,  82, 100])>
```

In [94]:

```
# Create the same function and decorate it with tf.function
@tf.function
def tf_function(x, y):
    return x ** 2 + y

tf_function(x, y)
```

Out[94]:

```
<tf.Tensor: shape=(10,), dtype=int32, numpy=array([ 10,  12,  16,  22,  30,  40,  52,  6
6,  82, 100])>
```

If you noticed no difference between the above two functions (the decorated one and the non-decorated one) you'd be right.

Much of the difference happens behind the scenes. One of the main ones being potential code speed-ups where possible.

Finding access to GPUs

We've mentioned GPUs plenty of times throughout this notebook.

So how do you check if you've got one available?

You can check if you've got access to a GPU using `tf.config.list_physical_devices()` .

In [95]:

```
print(tf.config.list_physical_devices())
```

```
[PhysicalDevice(name='/physical_device:CPU:0', device_type='CPU')]
```

If the above outputs an empty array (or nothing), it means you don't have access to a GPU (or at least TensorFlow can't find it).

If you're running in Google Colab, you can access a GPU by going to *Runtime -> Change Runtime Type -> Select GPU* (**note:** after doing this your notebook will restart and any variables you've saved will be lost).

Once you've changed your runtime type, run the cell below.

In [96]:

```
import tensorflow as tf
print(tf.config.list_physical_devices())
```

```
[PhysicalDevice(name='/physical_device:CPU:0', device_type='CPU')]
```

If you've got access to a GPU, the cell above should output something like:

```
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

You can also find information about your GPU using `!nvidia-smi` .

In [97]:

```
!nvidia-smi
```

Tue Dec 7 23:07:58 2021

NVIDIA-SMI 471.96			Driver Version: 471.96		CUDA Version: 11.4		
GPU	Name	TCC/WDDM	Bus-Id	Disp.A	Volatile	Uncorr.	ECC
Fan	Temp	Perf	Pwr:Usage/Cap		Memory-Usage	GPU-Util	Compute M.
						MIG M.	
0	NVIDIA GeForce ...	WDDM	00000000:01:00.0	Off			N/A
N/A	43C	P8	7W / N/A		117MiB / 4096MiB	4%	Default
							N/A

Processes:					
GPU	GI	CI	PID	Type	Process name
ID	ID				GPU Memory Usage
0	N/A	N/A	5096	C+G	...n64\EpicGamesLauncher.exe N/A

 **Note:** If you have access to a GPU, TensorFlow will automatically use it whenever possible.

Exercises

1. Create a vector, scalar, matrix and tensor with values of your choosing using `tf.constant()` .
2. Find the shape, rank and size of the tensors you created in 1.
3. Create two tensors containing random values between 0 and 1 with shape [5, 300] .
4. Multiply the two tensors you created in 3 using matrix multiplication.
5. Multiply the two tensors you created in 3 using dot product.
6. Create a tensor with random values between 0 and 1 with shape [224, 224, 3] .
7. Find the min and max values of the tensor you created in 6.
8. Created a tensor with random values of shape [1, 224, 224, 3] then squeeze it to change the shape to [224, 224, 3] .
9. Create a tensor with shape [10] using your own choice of values, then find the index which has the maximum value.
10. One-hot encode the tensor you created in 9.

Extensions

- Read through the [list of TensorFlow Python APIs](#), pick one we haven't gone through in this notebook, reverse engineer it (write out the documentation code for yourself) and figure out what it does.
- Try to create a series of tensor functions to calculate your most recent grocery bill (it's okay if you don't use the names of the items, just the price in numerical form).
 - How would you calculate your grocery bill for the month and for the year using tensors?
- Go through the [TensorFlow 2.x quick start for beginners](#) tutorial (be sure to type out all of the code yourself, even if you don't understand it).
 - Are there any functions we used in here that match what's used in there? Which are the same? Which haven't you seen before?
- Watch the video "[What's a tensor?](#)" - a great visual introduction to many of the concepts we've covered in this notebook.